AFIT/EN/TR/93-07

OOK-A Automated Knowledge System (OAKS) Code

Nancy L. Crowley, Major, USAF

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/EN/TR/93-07

AFIT/EN/TR/93-07

OORA Automated Knowledge System (OAKS) Code

Nancy L. Crowley, Major, USAF

AFIT/EN/TR/93-03

OORA Automated Knowledge System (OAKS) Code

September 1993

Nancy L. Crowley, Majhor, USAF
PhD Student, Department of Electrical and Computer Engineering
Air Force Institute of Technology
Wright-Patterson Air Force Base, Ohio 45433

OORA Automated Knowledge System (OAKS) Code

This technical rcport contains the code for an Air Force Institute of Technology Dissertation numbered AFIT/DS/ENG/93-11, dated September 1993. The dissertation investigated the use of an automated system to help capture object-oriented requirements. The code is written in Common Lisp and LISPView and was run on a SUN SPARCstation 2.

The correct capture of user requirements is an essential and difficult first step in software development. One method that aids in this process is object-oriented requirements analysis (OORA). This process makes use of method and domain knowledge to develop an object-oriented requirements specification. The research developed a object-oriented model that could be used as a basis for an automated system. An automated system, called the OORA Automated Knowledge System (OAKS), was developed. OAKS assists in the development of an object-oriented specification through the use of domain knowledge and knowledge of the structure of an object-oriented requirements specifications and relationships between its components.

The OAKS code was developed in six files. These files are presented in the following order:

1. "oaksd.lisp" This file contains the structure for the domain model and the domain model itself.

2. "oaksno.lisp" This file contains the LISP procedures which evaluated the domain model and the evolving user model.

3. "oaksmod.lisp" This file contains the LISP procedures which modified the domain model to rpoduce the problem model.

4. "oaksave.lisp" This file contains the LISP procedures which saved the user changes to the domain and problem models.

5. "oaksui.lisp" This file used LISPView to create a windowed user interface to OAKS.

6. "oaks.lisp" This file was used to start an OAKS session by bringing in the above files to create the OAKS environment.

OAKSD.LISP

The domain model

```
(in-package 'oaks)

;; Define a generic class.
;; This class is used as a basis for all classes in the domain model.
(clos:defclass generic-class ()
              ((name :initarg :name
                     :accessor name
                     :documentation "The name of the class")
                (description :initarg :desc
                             :accessor desc
                             :documentation "A description of the class")
                (state-space :initarg :state-space
                             :accessor state-space
                             :documentation "The class state space")
                (services :initarg :services
                          :accessor services
                          :documentation "The class services")
                (inheritance :initarg :inheritance
                             :accessor inheritance
                             :documentation "The immediate superclasses")
                (whole-part :initarg :whole-part
                            :accessor whole-part
                            :documentation "The whole-part relation")
                (relationships :initarg :relation
                               :accessor relation
                               :documentation "Other relationships")
                (need-verified :initarg :verif
                               :accessor verif
                               :documentation
                               "Does the class need user verification"))
              (:documentation "A generic class"))

;; The sets that can be used as a base for an attribute are
;; integers (int), reals (real), characters (char), strings
;; (str), enumerated types (enum), lists of elements, an
;; instance of a class (class), the base of another attribute,
;; or a list of elements.  A list of elements will be lists
;; of the "attrs" structure.  If it is an instance of a
;; class, the class name is the lower value.  If it is the same
;; set of an attribute of another class, the base set is "attrib",
;; the lower value is the class name and the upper value is the
;; attribute name.
(defun proper-attr-setp (a-set)
 (or
  (eql a-set '())
  (eql a-set 'enum)
  (eql a-set 'int)
  (eql a-set 'real)
  (eql a-set 'char)
  (eql a-set 'str)
  (eql a-set 'bool)
  (eql a-set 'class)
  (eql a-set 'attrib)
  (listp a-set)))

(deftype legal-set ()
 '(satisfies proper-attr-setp))

;; Define a record structure that is used as the form for the set of
;; each attribute.
(defstruct attrs
 (base 'int :type legal-set)
 (lower 'none)
 (upper 'none))

;; Define a record structure that is used as the form of each
;; input and output parameters for the input and output sets
;; of services. Each input parameter must contain the parameter name
;; and the legal set of values for that parameter. If the legal set of
;; values is that of an attribute of another class, the set of values
;; is a list of the form (:a class-name attr-name). If the legal set
;; of values is the instances of a class, the set of values is a list
;; of the form (c: class-name). The services are modelled as functions
;; and return a single value. This value can be a single element or a
;; list of elements. Each ouput parameter is therefore defined soley
;; by its set of legal values and no name is required.
;; For example, if the input parameter, named ip, is of the same
;; set as attr-1, the record structure would be:
;;    (make-parameterf :name 'ip :values 'attr-1))
(defstruct parameterf
 (name '())
 values)

;; Define a record structure that is used as the form for each
;; attribute/value pair in the postconditions
;; of services.
(defstruct attr-val
 name
 value)

;; Define a record structure that is used as the form of the
;; postcondition for each service
(defstruct postf
 (atts '() :type list)
 ;; a list of attr-val that are the local attributes that changed
 (messages '() :type list))
 ;; a list of pairs consisting of a classname and a service name.
 ;; These are services of other classes that are used.

;; Define a record structure for one relation, either whole/part
;; or other relation. The whole is class 1. Range 1 and 2 are
;; lists of two elements. The first is the lower bound and the
;; second is the upper bound.
(defstruct relation
 (name 'whole/part)
 class1
 range1
 class2
 range2)

;; Define one attribute, which is a member of the set state-space.
(clos:defclass attribute ()
        ((name :initarg :name
               :initform "   "
               :accessor name
               :documentation "The name of the attribute")
          (description :initarg :desc
                       :accessor desc
                       :documentation "A description of the attribute")
          (initial-value :initarg :initial-value
                         :initform '()
                         :accessor initial-value
                         :documentation "Any initial value used when an object
                          of this class is created.")
          (a-set :initarg :a-set
                 :initform '()
                 :accessor a-set
                 :documentation "The set of legal values")
          (need-verified :initarg :verif
                         :accessor verif
                         :initform '()))
        (:documentation "A general structure for an attribute"))

;; Define one service, which is a member of the set of services
;; The form of the input-set:
;;   The input-set is a list that consists of the type parameterf
;; The form of the output-set:
;;   Same as the input-set.
;; The form of the preconditions:
;;   The preconditions will be a list that evaluates to true or false.
;; The form of the postconditions:
;;   The postconditions will be of the type post-f.
(clos:defclass service ()
        ((name :initarg :name
               :accessor name
               :documentation "The name of the service")
          (description :initarg :desc
                       :accessor desc
                       :documentation "A description of the service")
          (input-set :initarg :input-set
                     :initform '()
                     :accessor input-set
                     :documentation "The output parameter list")
          (output-set :initarg :output-set
                      :accessor output-set
                      :documentation "The output parameter list")
          (preconditions :initarg :pre
                         :accessor pre
                         :documentation "the preconditions")
          (postconditions :initarg :post
                          :accessor post
                          :documentation "The postconditions")
          (need-verified :initarg :verif
                         :accessor verif
                         :initform '()))
        (:documentation "A generic service class"))

;; Creates a class called squadron
;; To access each component of the list of attributes, use:
;;   (name (car (state-space system-mode)))
(setf squadron
  (let*
    ((name
      (make-instance 'attribute
                        :name 'name
                        :desc "The name of the squadron"
                        :a-set (make-attrs :base 'str) ))

     (flights
      (make-instance 'attribute
                        :name 'flights
                        :desc "A list of flights of the squadron"
                        :a-set (make-attrs :base `(,(make-attrs :base 'class
                                                                :lower 'flight))) ))

     (parking
      (make-instance 'attribute
                        :name 'parking
                        :desc "Aircraft parking, including spots on the runway and hangar slots."
                        :a-set (make-attrs :base 'class
                                           :lower 'aircraft-parking) ))

     (aircrew
      (make-instance 'attribute
                        :name 'aircrew
                        :desc "A list of all aircrew in the squadron"
                        :a-set (make-attrs :base `(,(make-attrs :base 'class
                                                                :lower 'aircrew))) ))

     (personnel
      (make-instance 'attribute
                        :name 'personnel
                        :desc "A list of all support personnel in the squadron"
                        :a-set (make-attrs :base `(,(make-attrs :base 'class
                                                                :lower 'support-person))) ))

     (change-name
      (make-instance 'service
                        :name 'change-name
                        :desc "Change the name of the squadron."
                        :input-set `(,(make-parameterf :name 'new-name
                                                       :values 'name))
                        :output-set '()
                        :pre '()
                        :post
                        (make-postf :atts `(,(make-attr-val
                                              :name 'name
                                              :value 'new-name))) ))

     (add-flight
      (make-instance 'service
                        :name 'add-flight
                        :desc "Add a flight to the squadron"
                        :input-set `(,(make-parameterf :name 'new-flight
                                                :values '(:c flight)))
                       :output-set '()
                       :pre '(not (member new-flight flights))
                       :post
                       (make-postf :atts `(,(make-attr-val
                                              :name 'flights
                                              :value '(cons new-flight flights)))
                                   :messages '((flight create))) ))

(remove-flight
 (make-instance 'service
                       :name 'remove-flight
                       :desc "Remove a flight from a squadron."
                       :input-set `(,(make-parameterf :name 'old-flight
                                                :values '(:c flight)))
                       :output-set '()
                       :pre '(member old-flight flights)
                       :post
                       (make-postf :atts `(,(make-attr-val
                                              :name 'flights
                                              :value '(delete old-flight flights)))
                                   :messages '((flight delete))) ))

(add-aircrew
 (make-instance 'service
                       :name 'add-aircrew
                       :desc "Add an aircrew member to the squadron."
                       :input-set `(,(make-parameterf :name 'new-person
                                                :values '(:c aircrew)))
                       :output-set '()
                       :pre '(not (member new-person aircrew))
                       :post
                       (make-postf :atts `(,(make-attr-val
                                              :name 'aircrew
                                              :value '(cons new-person aircrew)))
                                   :messages '((aircrew create))) ))

(remove-aircrew
 (make-instance 'service
                       :name 'remove-aircrew
                       :desc "Remove an aircrew member from the squadron."
                       :input-set `(,(make-parameterf :name 'old-person
                                                :values '(:c aircrew)))
                       :output-set '()
                       :pre '(member old-person aircrew)
                       :post
                       (make-postf :atts `(,(make-attr-val
                                              :name 'aircrew
                                              :value '(delete old-person aircrew)))
                                   :messages '((aircrew delete)) ) ))

(add-support
 (make-instance 'service
                       :name 'add-support
                       :desc "Add a support person to the squadron."
                       :input-set `(,(make-parameterf :name 'new-person
                                                  :values '(:c support-person)))
                        :output-set '()
                        :pre '(not (member new-person personnel))
                        :post
                        (make-postf :atts `(,(make-attr-val
                                              :name 'personnel
                                              :value '(cons new-person personnel)))
                                    :messages '((support-person create)) ) ))

   (remove-support
    (make-instance 'service
                        :name 'remove-support
                        :desc "Remove a support person from the squadron."
                        :input-set `(,(make-parameterf :name 'old-person
                                                  :values '(:c support-person)))
                        :output-set '()
                        :pre '(member old-person personnel)
                        :post
                        (make-postf :atts `(,(make-attr-val
                                              :name 'personnel
                                              :value '(delete old-person personnel)))
                                    :messages '((support-person delete))) )) )

   (make-instance 'generic-class
                       :name 'squadron
                       :desc "This class represents a squadron of planes.  Each squadron
consists of a number of flights, hangars and flight line spots for aircraft
parking, aircrew, and support personnel."
                       :state-space (list name flights parking aircrew personnel)
                       :services (list change-name add-flight remove-flight
                                   add-aircrew remove-aircrew
                                   add-support remove-support)
                       :inheritance '()
                       :whole-part `(,(make-relation :class1 'squadron
                                                  :range1 '(1 n)
                                                  :class2 'flight
                                                  :range2 '(1 1))
                                    ,(make-relation :class1 'squadron
                                                  :range1 '(1 1)
                                                  :class2 'aircraft-parking
                                                  :range2 '(1 1))
                                    ,(make-relation :class1 'squadron
                                                  :range1 '(1 n)
                                                  :class2 'aircrew
                                                  :range2 '(1 1))
                                    ,(make-relation :class1 'squadron
                                                  :range1 '(1 n)
                                                  :class2 'support-person
                                                  :range2 '(1 1)))
                       :relation '()
                       :verif '() )))

(setf flight
  (let*
    ((name
       (make-instance 'attribute
                    :name 'name
                    :desc "The name of the flight."
                    :a-set (make-attrs :base 'str)))

(type-aircraft
 (make-instance 'attribute
                    :name 'type-aircraft
                    :desc "The type of aircraft, such as F-16."
                    :a-set (make-attrs :base 'str) ))

(the-aircraft
 (make-instance 'attribute
                    :name 'the-aircraft
                    :desc "A list of the aircraft in the flight."
                    :a-set (make-attrs :base `(,(make-attrs :base 'class
                                                                 :lower 'aircraft))) ))

(squadron
 (make-instance 'attribute
                    :name 'squadron
                    :desc "The squadron this flight is a part of."
                    :a-set (make-attrs :base 'class
                                              :lower 'squadron) ))

(change-name
 (make-instance 'service
                    :name 'change-name
                    :desc "Changes the name of the flight."
                    :input-set `(,(make-parameterf :name 'new-name
                                                          :values 'name))
                    :output-set '()
                    :pre '()
                    :post
                    (make-postf :atts `(,(make-attr-val
                                               :name 'name
                                               :value 'new-name))) ))

(change-type-aircraft
 (make-instance 'service
                    :name 'change-type-aircraft
                    :desc "Change the type of aircraft in the flight."
                    :input-set `(,(make-parameterf :name 'new-type
                                                          :values 'type-aircraft))
                    :output-set '()
                    :pre '()
                    :post
                    (make-postf :atts `(,(make-attr-val
                                               :name 'type-aircraft
                                               :value 'new-type))) ))

(add-aircraft
 (make-instance 'service
                    :name 'add-aircraft
                    :desc "Add an aircraft to the flight."
                    :input-set `(,(make-parameterf :name 'new-ac
                                                          :values '(:c aircraft)))
```

```
                        :output-set '()
                        :pre '(not (member new-ac the-aircraft))
                        :post
                        (make-postf :atts `(,(make-attr-val
                                               :name 'the-aircraft
                                               :value '(cons new-ac the-aircraft)))
                                    :messages '((aircraft create)) ) ))

      (remove-aircraft
       (make-instance 'service
                        :name 'remove-aircraft
                        :desc "Remove an aircraft from the flight."
                        :input-set `(,(make-parameterf :name 'old-ac
                                                        :values '(:c aircraft)))
                        :output-set '()
                        :pre '(member old-ac the-aircraft)
                        :post
                        (make-postf :atts `(,(make-attr-val
                                               :name 'the-aircraft
                                               :value '(delete old-ac the-aircraft)))
                                    :messages '((aircraft delete)) ) ))
      (aircraft-list
       (make-instance 'service
                        :name 'aircraft-list
                        :desc "Return a list of the aircraft in the squadron."
                        :input-set '()
                        :output-set `(,(make-parameterf :values 'the-aircraft))
                        :pre '()
                        :post '() )) )

    (make-instance 'generic-class
                   :name 'flight
                   :desc "The class represents a flight of planes.  It contains information on the
type of aircraft in the flight and a list of the aircraft."
                   :state-space (list name type-aircraft the-aircraft
                                      squadron)
                   :services (list change-name change-type-aircraft
                                   add-aircraft remove-aircraft aircraft-list)
                   :inheritance '()
                   :whole-part `(,(make-relation :class1 'squadron
                                                  :range1 '(1 n)
                                                  :class2 'flight
                                                  :range2 '(1 1))
                                 ,(make-relation :class1 'flight
                                                  :range1 '(1 n)
                                                  :class2 'aircraft
                                                  :range2 '(1 1)) )
                   :relation '()
                   :verif '()) ))

(setf aircraft
 (let*
   ((model-number
      (make-instance 'attribute
                     :name 'model-number
                     :desc "The model number of the aircraft."
                   :a-set (make-attrs :base 'str) ))

(tail-number
   (make-instance 'attribute
                   :name 'tail-number
                   :desc "The tail number of the aircraft."
                   :a-set (make-attrs :base 'int
                                      :lower 1
                                      :upper 9999) ))

(the-flight
   (make-instance 'attribute
                   :name 'the-flight
                   :desc "The flight the aircraft belongs to."
                   :a-set (make-attrs :base 'class
                                      :lower 'flight) ))

(status
   (make-instance 'attribute
                   :name 'status
                   :desc "The status of the aircraft."
                   :a-set (make-attrs :base 'enum
                                      :lower '(fully-mission-capable
                                               partly-mission-capable
                                               not-mission-capable)) ))

(inop-parts
   (make-instance 'attribute
                   :name 'inop-parts
                   :desc "A list of inoperative parts of this aircraft."
                   :a-set (make-attrs :base `(,(make-attrs :base 'class
                                                           :lower 'aircraft-part))) ))

(schedule
   (make-instance 'attribute
                   :name 'schedule
                   :desc "The flight schedule for the aircraft."
                   :a-set (make-attrs :base 'class
                                      :lower 'aircraft-schedule) ))

(configuration
   (make-instance 'attribute
                   :name 'configuration
                   :desc "Lists of parts of the aircraft."
                   :a-set (make-attrs :base `(,(make-attrs :base 'class
                                                           :lower 'aircraft-part))) ))

(in-op-part
   (make-instance 'service
                   :name 'in-op-part
                   :desc "Add the part to the list of inoperative parts
                   and possibly change the status of the aircraft."
                   :input-set `(,(make-parameterf :name 'ap
                                                  :values '(:c aircraft-part)))
                   :output-set '()
                   :pre '(member ap configuration)
```

```
                    :post
                    (make-postf :atts `(,(make-attr-val
                                         :name 'inop-parts
                                         :value '(cons ap inop-parts))
                                        ,(make-attr-val
                                         :name 'status
                                         :value 'changed) )) ))

(op-part
  (make-instance 'service
                    :name 'op-part
                    :desc "Remove the part form the list of inoperative
                    parts and possibly change the staus of the aircraft."
                    :input-set `(,(make-parameterf :name 'ap
                                              :values '(:c aircraft-part)))
                    :output-set '()
                    :pre '(member ap inop-parts)
                    :post
                    (make-postf :atts `(,(make-attr-val
                                         :name 'inop-parts
                                         :value '(delete ap inop-parts))
                                        ,(make-attr-val
                                         :name 'status
                                         :value 'changed) )) ))

(get-tail-number
  (make-instance 'service
                    :name 'get-tail-number
                    :desc "Return the tail number of the aircraft."
                    :input-set '()
                    :output-set `(,(make-parameterf :values 'tail-number))
                    :pre '()
                    :post '() ))

(get-flight
  (make-instance 'service
                    :name 'get-flight
                    :desc "Return the flight the aircraft belongs to."
                    :input-set '()
                    :output-set `(,(make-parameterf :values 'the-flight))
                    :pre '()
                    :post '() ))

(get-status
  (make-instance 'service
                    :name 'get-status
                    :desc "Return the status of the aircraft."
                    :input-set '()
                    :output-set `(,(make-parameterf :values 'status))
                    :pre '()
                    :post '() ))

(get-config
  (make-instance 'service
                    :name 'get-config
                    :desc "Return the list of parts for the aircraft."
                    :input-set '()
                    :output-set `(,(make-parameterf :values 'configuration))
                    :pre '()
                    :post '() ))

   (add-part
      (make-instance 'service
                    :name 'add-part
                    :desc "Add a part to the aircraft."
                    :input-set `(,(make-parameterf :name 'new-part
                                                   :values '(:c aircraft-part)))
                    :output-set '()
                    :pre '(not (member new-part configuration))
                    :post
                    (make-postf :atts `(,(make-attr-val
                                          :name 'configuration
                                          :value '(cons new-part configuration)))
                                :messages '((aircraft-part create)) ) ))

   (remove-part
      (make-instance 'service
                    :name 'remove-part
                    :desc "Remove a part permanently from the aircraft."
                    :input-set `(,(make-parameterf :name 'old-part
                                                   :values '(:c aircraft-part)))
                    :output-set '()
                    :pre '(member old-part configuration)
                    :post
                    (make-postf :atts `(,(make-attr-val
                                          :name 'configuration
                                          :value '(delete old-part configuration)))
                                :messages '((aircraft delete)) ) ))


   (get-sched
      (make-instance 'service
                    :name 'get-sched
                    :desc "Return the flight schedule for the aircraft."
                    :input-set '()
                    :output-set `(,(make-parameterf :values 'schedule))
                    :pre '()
                    :post '() )) )

   (make-instance 'generic-class
                  :name 'aircraft
                  :desc "The class represents one aircraft. It contains information
on the parts on the aircraft, the status of the aircraft, the flight schedule, and any
inoperative parts. When an inoperative part is added, the status of the aircraft
may change."
                  :state-space (list model-number tail-number the-flight
                                     status inop-parts schedule configuration)
                  :services (list in-op-part op-part get-tail-number
                                  get-flight get-status add-part remove-part
                                  get-sched get-config)
                  :inheritance '()
                  :whole-part `(,(make-relation :class1 'flight
                                             :range1 '(1 n)
                                             :class2 'aircraft
                                             :range2 '(1 1))
                               ,(make-relation :class1 'aircraft
                                             :range1 '(1 n)
                                             :class2 'aircraft-part
                                             :range2 '(1 1)) )
                     :relation `(,(make-relation :name 'has-a/for-a
                                             :class1 'aircraft
                                             :range1 '(1 1)
                                             :class2 'aircraft-schedule
                                             :range2 '(1 1)))
                     :verif '()) ))

(setf aircraft-part
  (let*
    ((part-name
        (make-instance 'attribute
                        :name 'part-name
                        :desc "The name of the part."
                        :a-set (make-attrs :base 'enum
                                           :lower '(part-a part-b part-c) )))

       (a-aircraft
        (make-instance 'attribute
                        :name 'a-aircraft
                        :desc "The aircraft the part belongs to."
                        :a-set (make-attrs :base 'class
                                           :lower 'aircraft) ))

       (number-of-flight-hours
        (make-instance 'attribute
                        :name 'number-of-flight-hours
                        :desc "The number of flight hours on the part."
                        :a-set (make-attrs :base 'int
                                           :lower 0) ))

       (repair-shop
        (make-instance 'attribute
                        :name 'repair-shop
                        :desc "The repair shop that repairs this part."
                        :a-set (make-attrs :base 'class
                                           :lower 'support-shop) ))
       (current-symptoms
        (make-instance 'attribute
                        :name 'current-symptoms
                        :desc "The current symptoms for a non-operative part."
                        :a-set (make-attrs :base 'attrib
                                           :lower 'repair-symptoms
                                           :upper 'legal-symptoms-list) ))

       (status
        (make-instance 'attribute
                        :name 'status
                        :desc "Whether the part in operative or needs repair."
                        :a-set (make-attrs :base 'enum
                                    :lower '(operative need-repair)) ))

(symptom-analysis
 (make-instance 'attribute
                :name 'symptom-analysis
                :desc "The legal repair symptoms for this part."
                :a-set (make-attrs :base 'class
                                    :lower 'repair-symptoms) ))

(history
 (make-instance 'attribute
                :name 'history
                :desc "The history of repairs made to this part."
                :a-set (make-attrs :base 'class
                                    :lower 'maintenance-history) ))

(periodic
 (make-instance 'attribute
                :name 'periodic
                :desc "The periodic maintenance required for this part."
                :a-set (make-attrs :base 'class
                                    :lower 'periodic-maintenance) ))

(get-aircraft
 (make-instance 'service
                :name 'get-aircraft
                :desc "Return the aircraft the part belongs to."
                :input-set '()
                :output-set `(,(make-parameterf :values 'a-aircraft))
                :pre '()
                :post '() ))

(get-part-name
 (make-instance 'service
                :name 'get-part-name
                :desc "Return the name of the part."
                :input-set '()
                :output-set `(,(make-parameterf :values 'part-name))
                :pre '()
                :post '() ))

(get-sym-analysis
 (make-instance 'service
                :name 'get-sym-analysis
                :desc "Return the repair symptoms object for this part."
                :input-set '()
                :output-set `(,(make-parameterf :values 'symptom-analysis))
                :pre '()
                :post '() ))

(inoperative
 (make-instance 'service
                :name 'inoperative
                :desc "Change the status of the part to inoperative."
                :input-set `(,(make-parameterf :name 'symptoms
                                                :values
```

```
                                     '(:a repair-symptoms legal-symptoms-list)))
                          :output-set '()
                          :pre '()
                          :post
                          (make-postf :atts `(,(make-attr-val
                                                  :name 'current-symptoms
                                                  :value
                                                  '(cons symptoms current-symptoms))
                                                 ,(make-attr-val
                                                  :name 'status
                                                  :value 'need-repair))
                                      :messages '((support-shop schedule-repair)
                                                  (aircraft in-op-part))) ))

(repaired
 (make-instance 'service
                          :name 'repaired
                          :desc "Change the status of the part to operative."
                          :input-set `(,(make-parameterf :name 'type
                                                         :values
                                                         '(:a maintenance-history type)))
                          :output-set '()
                          :pre '()
                          :post
                          (make-postf :atts `(,(make-attr-val
                                                  :name 'current-symptoms
                                                  :value '())
                                                 ,(make-attr-val
                                                  :name 'status
                                                  :value 'operative))
                                      :messages '((aircraft op-part)
                                                  (maintenance-history add-maintenance))) ))

(update-flight-hours
 (make-instance 'service
                          :name 'update-flight-hours
                          :desc "Increase the number of flight hours on the part."
                          :input-set `(,(make-parameterf :name 'new-hours
                                                         :values 'int))
                          :output-set '()
                          :pre '()
                          :post
                          (make-postf :atts `(,(make-attr-val
                                                  :name 'number-of-flight-hours
                                                  :value '(+ number-of-flight-hours new-hours)))
                                      :messages '((periodic-maintenance check-list)) ) ))

(current-symptoms-list
 (make-instance 'service
                          :name 'current-symptoms-list
                          :desc "Return the current list of symptoms."
                          :input-set '()
                          :output-set `(,(make-parameterf :values 'current-symptoms))
                          :pre '()
                          :post '() ))
```

```
        (shop-name
         (make-instance 'service
                        :name 'shop-name
                        :desc "Return the name of the shop that repairs this part."
                        :input-set '()
                        :output-set `(,(make-parameterf :values 'repair-shop))
                        :pre '()
                        :post '() )) )

    (make-instance 'generic-class
                   :name 'aircraft-part
                   :desc "This class represents one part of an aircraft.  An
 instance of this class will be created for each part on each aircraft.
Therefore, each instance will represent a part of a particular aircraft.  The
information that needs to be kept are the aircraft the part belongs to, the
number of flight hours on the part, the repair shop for the part, teh status and
history of the part, and the periodic maintenance for te part.  When the hours are
changed, periodic maintenance may be scheduled.  When the part is inoperative,
a message is sent to the aircraft because the aircraft status may change."
                   :state-space (list part-name a-aircraft
                                      number-of-flight-hours
                                      repair-shop current-symptoms
                                      status symptom-analysis history
                                      periodic)
                   :services (list get-aircraft get-part-name
                                   inoperative repaired
                                   update-flight-hours get-sym-analysis
                                   current-symptoms-list shop-name)
                   :inheritance '()
                   :whole-part `(,(make-relation :class1 'aircraft
                                                 :range1 '(1 n)
                                                 :class2 'aircraft-part
                                                 :range2 '(1 1)))
                   :relation `(,(make-relation :name 'fixed-by/fixes
                                               :class1 'aircraft-part
                                               :range1 '(1 1)
                                               :class2 'support-shop
                                               :range2 '(1 n))
                               ,(make-relation :name 'has/for-a
                                               :class1 'aircraft-part
                                               :range1 '(1 1)
                                               :class2 'repair-symptoms
                                               :range2 '(1 1))
                               ,(make-relation :name 'has/of-an
                                               :class1 'aircraft-part
                                               :range1 '(1 1)
                                               :class2 'maintenance-history
                                               :range2 '(1 1))
                               ,(make-relation :name 'requires/for
                                               :class1 'aircraft-part
                                               :range1 '(1 1)
                                               :class2 'periodic-maintenance
                                               :range2 '(1 n)))
                   :verif '()) ))

(setf periodic-task
(let*
  ((part-name
     (make-instance 'attribute
                    :name 'part-name
                    :desc "The part that the periodic tasks apply to."
                    :a-set (make-attrs :base 'attrib
                                       :lower 'aircraft-part
                                       :upper 'part-name) ))

    (hours
     (make-instance 'attribute
                    :name 'hours
                    :desc "The number of hours before this periodic
                    maintenance is required."
                    :a-set (make-attrs :base 'int) ))

    (hangar-required
     (make-instance 'attribute
                    :name 'hangar-required
                    :desc "Whether a hangar is required for the task."
                    :a-set (make-attrs :base 'bool) ))

    (task-name
     (make-instance 'attribute
                    :name 'task-name
                    :desc "The name of the periodic task."
                    :a-set (make-attrs :base 'str) ))

    (hours-to-task
     (make-instance 'service
                    :name 'hours-to-task
                    :desc "The number of hours the task requires."
                    :input-set '()
                    :output-set `(,(make-parameterf :values 'hours))
                    :pre '()
                    :post '() ))

    (change-time-before-repair
     (make-instance 'service
                    :name 'change-time-before-repair
                    :desc "Change the number of hours before the
                    periodic task is needed."
                    :input-set `(,(make-parameterf :name 'new-hours
                                                   :values 'int))
                    :output-set '()
                    :pre '()
                    :post
                    (make-postf :atts `(,(make-attr-val
                                          :name 'hours
                                          :value 'new-hours))) ))

    (hangar-needed
     (make-instance 'service
                    :name 'hangar-needed
                    :desc "Whether or not a hangar is neede for the work."
                    :input-set '()
```

```
                         :output-set `(,(make-parameterf :values 'hangar-required))
                         :pre '()
                         :post '() )) )

     (make-instance 'generic-class
                    :name 'periodic-task
                    :desc "This class represents a periodic task for an aircraft part.
It maintains information on the part it applies to, the hours when the task is required,
whether a hangar is required for the task, and the length of time to perform the
task."
                    :state-space (list part-name hours hangar-required
                                      task-name)
                    :services (list hours-to-task change-time-before-repair
                                   hangar-needed)
                    :inheritance '()
                    :whole-part `(,(make-relation :class1 'periodic-maintenance
                                                  :range1 '(1 n)
                                                  :class2 'periodic-task
                                                  :range2 '(1 1)))
                    :relation '()
                    :verif '()) ))

(setf periodic-maintenance
  (let*
     ((part
        (make-instance 'attribute
                         :name 'part
                         :desc "The part this periodic maintenance applies to."
                         :a-set (make-attrs :base 'attrib
                                           :lower 'aircraft-part
                                           :upper 'part-name) ))

       (task-list
        (make-instance 'attribute
                         :name 'task-list
                         :desc "A list of all periodic tasks required for a part."
                         :a-set (make-attrs :base `(,(make-attrs :base 'class
                                                                :lower 'periodic-task))) ))

       (check-list
        (make-instance 'service
                         :name 'check-list
                         :desc "Checks to see if periodic maintenance is
                         required based on a new number of flight hours
                         for a part."
                         :input-set `(,(make-parameterf :name 'new-hours
                                                       :values 'int))
                         :output-set '()
                         :pre '()
                         :post
                         (make-postf :messages '((periodic-task hours-to-task)
                                                (aircraft-part shop-name)
                                                (support-shop schedule-periodic))) )) )

     (make-instance 'generic-class
                    :name 'periodic-maintenance
                    :desc "This class represents all periodic maintenance required for
a part.  It consists of a list of periodic tasks.  Each time hours are added to a
part, the class checks to see if any periodic tasks are required."
                    :state-space (list part task-list)
                    :services (list check-list)
                    :inheritance '()
                    :whole-part `(,(make-relation :class1 'periodic-maintenance
                                                   :range1 '(1 n)
                                                   :class2 'periodic-task
                                                   :range2 '(1 1)))
                    :relation `(,(make-relation :name 'requires/for
                                                 :class1 'aircraft-part
                                                 :range1 '(1 1)
                                                 :class2 'periodic-maintenance
                                                 :range2 '(1 1)))
                    :verif '()) ))

(setf repair-symptoms
  (let*
    ((part-name
        (make-instance 'attribute
                       :name 'part-name
                     :desc "The name of the part the repair symptoms apply to."
                       :a-set (make-attrs :base 'attrib
                                          :lower 'aircraft-part
                                          :upper 'part-name) ))

       (legal-symptoms-list
        (make-instance 'attribute
                       :name 'legal-symptoms-list
                       :desc "The list of legal symptoms for a part."
                       :a-set (make-attrs :base 'enum) ))

       (determine-hangar-need
        (make-instance 'service
                       :name 'determine-hangar-need
                       :desc "Returns true if a hangar is needed based on
                       the current symptoms."
                       :input-set `(,(make-parameterf :name 'sym
                                                      :values
                                                      'legal-symptoms-list))
                       :output-set `(,(make-parameterf :values 'bool))
                       :pre '()
                       :post '() )) )

    (make-instance 'generic-class
                    :name 'repair-symptoms
                    :desc "The class represents the legal set of repair symptoms
for a part.  It determines if a hangar is needed based on the current set of
symptoms."
                    :state-space (list part-name legal-symptoms-list)
                    :services (list determine-hangar-need)
                    :inheritance '()
                    :whole-part '()
                    :relation `(,(make-relation :name 'has/for-a
                                                 :class1 'aircraft-part
```

```
                                        :range1 '(1 1)
                                        :class2 'repair-symptoms
                                        :range2 '(1 1)))
                    :verif '()) ))

(setf maintenance-history
  (let*
    ((part
        (make-instance 'attribute
                       :name 'part
                       :desc "The part the maintenance history is on."
                       :a-set (make-attrs :base 'class
                                          :lower 'aircraft-part) ))

       (type
        (make-instance 'attribute
                       :name 'type
                       :desc "The type of maintenance."
                       :a-set (make-attrs :base 'attrib
                                          :lower 'support-shop
                                          :upper 'type) ))

       (history-list
        (make-instance 'attribute
                       :name 'history-list
                       :desc "History of the maintenance on the part."
                       :a-set (make-attrs :base `(,(make-attrs :base 'attrib
                                                               :lower 'schedule-event
                                                               :upper 'day)
                                                  ,(make-attrs :base 'attrib
                                                               :lower 'support-shop
                                                               :upper 'type))) ))

       (add-maintenance
        (make-instance 'service
                       :name 'add-maintenance
                       :desc "Add information to the maintenance history."
                       :input-set `(,(make-parameterf :name 'date
                                                      :values '(:a schedule-event day))
                                    ,(make-parameterf :name 'a-type
                                                      :values '(:a support-shop type)))
                       :output-set '()
                       :pre '()
                       :post
                       (make-postf :atts `(,(make-attr-val
                                             :name 'history-list
                                             :value '(cons '(date a-type)
                                                           history-list)))) )))

    (make-instance 'generic-class
                   :name 'maintenance-history
                   :desc "This class represents the maintenace history for a
particular part on one aircraft.  It keeps information on the type of maintenace
performed on the part."
                   :state-space (list part type history-list)
                   :services (list add-maintenance)
```

```
                    :inheritance '()
                    :whole-part '()
                    :relation `(,(make-relation :name 'has/of-an
                                                :class1 'aircraft-part
                                                :range1 '(1 1)
                                                :class2 'maintenance-history
                                                :range2 '(1 1)))
                    :verif '()) ))

(setf people
  (let*
    ((name
       (make-instance 'attribute
                      :name 'name
                      :desc "The name of the person."
                      :a-set (make-attrs :base 'str) ))

      (ssan
       (make-instance 'attribute
                      :name 'ssan
                      :desc "The person's social security number."
                      :a-set (make-attrs :base 'int
                                         :lower 0
                                         :upper 999999999) ))

      (squad
       (make-instance 'attribute
                      :name 'squad
                      :desc "the squadron the person is assigned to."
                      :a-set (make-attrs :base 'class
                                         :lower 'squadron) ))

      (afsc
       (make-instance 'attribute
                      :name 'afsc
                      :desc "The person's AF Specialty Code."
                      :a-set (make-attrs :base 'str) ))

      (change-squadron
       (make-instance 'service
                      :name 'change-squadron
                      :desc "Change the squadron the person belongs to."
                      :input-set `(,(make-parameterf :name 'new-squaoron
                                                      :values 'squad))
                      :output-set '()
                      :pre '()
                      :post
                      (make-postf :atts `(,(make-attr-val
                                            :name 'squad
                                            :value 'new-squadron))) )) )

    (make-instance 'generic-class
                    :name 'people
                    :desc "The class represents a person in the squadron.
Information on the person's name, social security number, AFSC and squadron are
kept."
```

```
                    :state-space (list name ssan squad afsc)
                    :services (list change-squadron)
                    :inheritance '()
                    :whole-part '()
                    :relation '()
                    :verif '()) ))

(setf support-person
  (let*
    ((shop
        (make-instance 'attribute
                        :name 'shop
                        :desc "The shop the supprt person works for."
                        :a-set (make-attrs :base 'class
                                           :lower 'support-shop) ))

     (type
      (make-instance 'attribute
                        :name 'type
                        :desc "The type of task the person is doing."
                        :a-set (make-attrs :base 'attrib
                                           :lower 'support-shop
                                           :upper 'type) ))

     (jobs-to-do
      (make-instance 'attribute
                        :name 'jobs-to-do
                        :desc "The pending jobs to be done."
                        :a-set (make-attrs :base `(,(make-attrs :base 'class
                                                                :lower 'aircraft-part)
                                                  ,(make-attrs :base 'attrib
                                                                :lower 'support-shop
                                                                :upper 'type))) ))

     (job-list
      (make-instance 'service
                        :name 'job-list
                        :desc "Return a lists of the jobs to be done."
                        :input-set '()
                        :output-set `(,(make-parameterf :values 'jobs-to-do))
                        :pre '()
                        :post '() ))

     (add-job
      (make-instance 'service
                        :name 'add-job
                        :desc "Add a job to the list of jobs to do."
                        :input-set `(,(make-parameterf :name 'new-job
                                                       :values '(:c aircraft-part))
                                    ,(make-parameterf :name 'the-type
                                                       :values 'type))
                        :output-set '()
                        :pre '()
                        :post
                        (make-postf :atts `(,(make-attr-val
                                              :name 'jobs-to-do
```

```
                                        :value '(cons (aircraft-part type)
                                                        jobs-to-do)))) ))

        (remove-job
         (make-instance 'service
                        :name 'remove-job
                        :desc "remove a job from the jobs to do list."
                        :input-set `(,(make-parameterf :name 'job
                                                       :values '(:c aircraft-part))
                                     ,(make-parameterf :name 'the-type
                                                       :values 'type))
                        :output-set '()
                        :pre '()
                        :post
                        (make-postf :atts `(,(make-attr-val
                                             :name 'jobs-to-do
                                             :value '(delete (job the-type)
                                                             jobs-to-do)))) )))

      (make-instance 'generic-class
                     :name 'support-person
                     :desc "The class represnts one support person in a squadron.
Information on the shop the person is assigned to, and a list of the jobs assigned
to this person are kept."
                     :state-space (list shop type jobs-to-do)
                     :services (list job-list add-job remove-job)
                     :inheritance '(people)
                     :whole-part `(,(make-relation :class1 'squadron
                                                   :range1 '(1 n)
                                                   :class2 'support-person
                                                   :range2 '(1 1)))
                     :relation '()
                     :verif '()) ))

(setf support-shop
  (let*
    ((list-of-people
        (make-instance 'attribute
                       :name 'list-of-people
                       :desc "A list of people assigned to the shop."
                       :a-set (make-attrs :base `(,(make-attrs :base 'class
                                                               :lower 'support-person))) ))

     (shop-name
      (make-instance 'attribute
                     :name 'shop-name
                     :desc "The name of the shop."
                     :a-set (make-attrs :base 'enum
                                        :lower '(a-shop b-shop c-shop
                                                 fuel-shop hydraulics
                                                 electrical-and-environmental
                                                 egress propulsion
                                                 machine-shop
                                                 corrosion-control
                                                 non-destructive-inspection
                                                 weapons PMEL LRU
```

```
                                            parachute
                                            flight-line-support
                                            aircraft-ground-equipment))))

(type
 (make-instance 'attribute
          :name 'type
          :desc "The type of a repair."
          :a-set (make-attrs :base 'enum
                             :lower '(fix periodic-task)) ))

(jobs-pending
 (make-instance 'attribute
          :name 'jobs-pending
          :desc "The jobs that are waiting for a hangar."
          :a-set (make-attrs :base `(,(make-attrs :base 'class
                                                  :lower 'aircraft-part)
                                     ,(make-attrs :base 'attrib
                                                  :lower 'support-shop
                                                  :upper 'type))) ))

(jobs-in-hangars
 (make-instance 'attribute
          :name 'jobs-in-hangars
          :desc "Jobs that are in a hangar."
          :a-set (make-attrs :base `(,(make-attrs :base 'class
                                                  :lower 'aircraft-part)
                                     ,(make-attrs :base 'class
                                                  :lower 'hangar)
                                     ,(make-attrs :base 'attrib
                                                  :lower 'support-shop
                                                  :upper 'type))) ))

(schedule-repair
 (make-instance 'service
          :name 'schedule-repair
          :desc "Schedule the repair of a part."
          :input-set `(,(make-parameterf :name 'ap
                                         :values '(:c aircraft-part)))
          :output-set '()
          :pre '()
          :post
          (make-postf :atts `(,(make-attr-val
                             :name 'jobs-pending
                             :value '(append jobs-pending
                                             '(ap fix))))
                 :messages '(
                  (aircraft-part get-sym-analysis)
                  (repair-symptoms determine-hangar-need)
                  (aircraft-part current-symptoms-list)
                  (aircraft-parking schedule-hangar)
                  (support-person job-list)
                  (support-person add-job))) ))

(hangar-available
 (make-instance 'service
                      :name 'hangar-available
                      :desc "Provides a hangar to be used for pending
                      maintenance."
                      :input-set `(,(make-parameterf :name 'han
                                                     :values '(:c hangar))
                                  ,(make-parameterf :name 'ap
                                                     :values '(:c aircraft-part))
                                  ,(make-parameterf :name 'a-type
                                                     :values 'type))
                      :output-set '()
                      :pre '(not (eql jobs-pending '()))
                      :post
                      (make-postf :atts `(,(make-attr-val
                                            :name 'jobs-pending
                                            :value
                                            '(delete ap jobs-pending))
                                           ,(make-attr-val
                                            :name 'jobs-in-hangars
                                            :value
                                            '(cons (ap han) jobs-in-hangars)))
                                  :messages '((support-person job-list)
                                              (support-person add-job))) ))

(schedule-periodic
 (make-instance 'service
                      :name 'schedule-periodic
                      :desc "Schedule periodic tasks."
                      :input-set `(,(make-parameterf :name 'ap
                                                     :values '(:c aircraft-part))
                                  ,(make-parameterf :name 'pt
                                                     :values '(:c periodic-task)))
                      :output-set '()
                      :pre '()
                      :post
                      (make-postf :atts `(,(make-attr-val
                                            :name 'jobs-pending
                                            :value
                                            '(append jobs-pending (ap pt))))
                                  :messages '(
                                   (aircraft-parking schedule-hangar)
                                   (periodic-task hangar-needed)
                                   (support-person job-list)
                                   (support-person add-job)) )))

(repair-complete
 (make-instance 'service
                      :name 'repair-complete
                      :desc "A repair is completed."
                      :input-set `(,(make-parameterf :name 'ap
                                                     :values '(:c aircraft-part))
                                  ,(make-parameterf :name 'the-type
                                                     :values 'type))
                      :output-set '()
                      :pre '()
                      :post
                      (make-postf :atts `(,(make-attr-val
                                        :name 'jobs-in-hangars
                                        :value
                                        '(delete (ap the-type)
                                                 jobs-in-hangars)))
                                  :messages '(
                                   (support-person job-list)
                                   (support-person remove-job)
                                   (aircraft-part repaired)
                                   (aircraft-parking release-hangar))) )))

   (make-instance 'generic-class
                    :name 'support-shop
                    :desc "The class represents one  support shop. The support
shop is responsible for the scheduling of support presonnel assigned to the shop,
and the repair of parts assigned to the shop. It keeps track of repair jobs that
are pending and those under repair in hangars. When a repair is needed, it schedules
the repair. When a repair is complete, it updates the aprt information and
releases personnel and facilities."
                    :state-space (list list-of-people shop-name
                                       type jobs-pending
                                       jobs-in-hangars)
                    :services (list schedule-repair hangar-available
                                    schedule-periodic repair-complete)
                    :inheritance '()
                    :whole-part '()
                    :relation `(,(make-relation :name 'fixed-by/fixes
                                                :class1 'aircraft-part
                                                :range1 '(1 1)
                                                :class2 'support-shop
                                                :range2 '(1 n)))
                    :verif '()) ))

(setf aircrew
  (let*
   ((type
     (make-instance 'attribute
                      :name 'type
                      :desc "The type of function the aircrew performs."
                      :a-set (make-attrs :base 'enum
                                         :lower '(pilot navigator EWO)) ))

     (aircraft-checked-out-in
      (make-instance 'attribute
                      :name 'aircraft-checked-out-in
                      :desc "The aircraft the aircrew is qualifed in."
                      :a-set (make-attrs :base 'class
                                         :lower 'aircraft) ))

     (hours
      (make-instance 'attribute
                      :name 'hours
                      :desc "The number of hours flown."
                      :a-set (make-attrs :base 'int
                                         :lower 0) ))

     (schedule
          (make-instance 'attribute
                         :name 'schedule
                         :desc "The aircrew member's schedule."
                         :a-set (make-attrs :base 'class
                                            :lower 'aircrew-schedule) ))

         (get-sched
          (make-instance 'service
                         :name 'get-sched
                         :desc "Returns the schedule for the aircrew member."
                         :input-set '()
                         :output-set `(,(make-parameterf :values 'schedule))
                         :pre '()
                         :post '() ))

         (update-hours
          (make-instance 'service
                         :name 'update-hours
                         :desc "Update the number of hours flown."
                         :input-set `(,(make-parameterf :name 'new-hours
                                                        :values 'hours))
                         :output-set '()
                         :pre '()
                         :post
                         (make-postf :atts `(,(make-attr-val
                                              :name 'hours
                                              :value '(+ hours new-hours)))) )) )

     (make-instance 'generic-class
                    :name 'aircrew
                    :desc "The class represents one aircrew member.  Information
on he type of the aircrew, the aircraft the aircrew is checked out on, the
hours flown, and the flight schedule are maintained."
                    :state-space (list type aircraft-checked-out-in
                                       hours schedule)
                    :services (list get-sched update-hours)
                    :inheritance '(people)
                    :whole-part `(,(make-relation :class1 'squadron
                                                  :range1 '(1 n)
                                                  :class2 'aircrew
                                                  :range2 '(1 1)))
                    :relation `(,(make-relation :name 'has/of
                                                :class1 'aircrew
                                                :range1 '(1 1)
                                                :class2 'aircrew-schedule
                                                :range2 '(1 1)))
                    :verif '()) ))

(setf mission
  (let*
    ((date
       (make-instance 'attribute
                      :name 'date
                      :desc "The date of the mission."
                      :a-set (make-attrs :base 'int) ))
```

```
(mission-type
 (make-instance 'attribute
                :name 'mission-type
                :desc "The type of mission."
                :a-set (make-attrs :base 'enum
                                   :lower '(test eval)) ))

(ac-info
 (make-instance 'attribute
                :name 'ac-info
                :desc "The aircraft needed, the time each
           aircraft is required, and the aircraft
           configuration."
                :a-set (make-attrs :base `(,(make-attrs :base 'class
                                                        :lower 'aircraft)
                                           ,(make-attrs :base 'attrib
                                                        :lower 'schedule-event
                                                        :upper 'duration)
                                           ,(make-attrs :base 'attrib
                                                        :lower 'aircraft
                                                        :upper 'configuration))) ))

(aircrew-list
 (make-instance 'attribute
                :name 'aircrew-list
                :desc "A list of the aircrew needed and the planes
           they are assigned to."
                :a-set (make-attrs :base `(,(make-attrs :base 'class
                                                        :lower 'aircrew)
                                           ,(make-attrs :base 'class
                                                        :lower 'aircraft))) ))

(time
 (make-instance 'attribute
                :name 'time
                :desc "The start time and duration of the mission."
                :a-set (make-attrs :base `(,(make-attrs :base 'attrib
                                                        :lower 'schedule-event
                                                        :upper 'start-time)
                                           ,(make-attrs :base 'attrib
                                                        :lower 'schedule-event
                                                        :upper 'duration))) ))

(range-info
 (make-instance 'attribute
                :name 'range-info
                :desc "Information on the range required for the mission."
                :a-set (make-attrs :base `(,(make-attrs :base 'real)
                                           ,(make-attrs :base 'int))) ))

(status
 (make-instance 'attribute
                :name 'status
                :desc "The status of the mission."
                :a-set (make-attrs :base 'enum
                                   :lower '(cancelled complete)) ))
```

```
(get-aircraft
 (make-instance 'service
                        :name 'get-aircraft
                        :desc "Get the aircraft involved in the mission."
                        :input-set '()
                        :output-set `(,(make-parameterf :values '((:c aircraft)) ))
                        :pre '()
                        :post '() ))

(get-duration
 (make-instance 'service
                        :name 'get-duration
                        :desc "return the duration of the mission."
                        :input-set '()
                        :output-set `(,(make-parameterf :values 'real))
                        :pre '()
                        :post '() ))

(get-date
 (make-instance 'service
                        :name 'get-date
                        :desc "return the date of the mission."
                        :input-set '()
                        :output-set `(,(make-parameterf :values 'date))
                        :pre '()
                        :post '() ))

(get-config
 (make-instance 'service
                        :name 'get-config
                        :desc "Return the configuration of an aircraft."
                        :input-set `(,(make-parameterf :name 'ac
                                                        :values '(:c aircraft)))
                        :output-set `(,(make-parameterf :values '(:a aircraft configuration)))
                        :pre '()
                        :post '() ))

(all-aircrew
 (make-instance 'service
                        :name 'all-aircrew
                        :desc "Return the aircrew list."
                        :input-set '()
                        :output-set `(,(make-parameterf :values 'aircrew-list))
                        :pre '()
                        :post '() ))

(get-mission-type
 (make-instance 'service
                        :name 'get-mission-type
                        :desc "Get the type of mission."
                        :input-set '()
                        :output-set `(,(make-parameterf :values 'mission-type))
                        :pre '()
                        :post '() ))
```

```
(get-range-info
 (make-instance 'service
                :name 'get-range-info
                :desc "Get the range information."
                :input-set '()
                :output-set `(,(make-parameterf :values 'range-info))
                :pre '()
                :post '() ))

(get-aircrew
 (make-instance 'service
                :name 'get-aircrew
                :desc "Return the aircrew on the mission."
                :input-set '()
                :output-set `(,(make-parameterf :values '((:c aircrew)) ))
                :pre '()
                :post '() ))

(change-date
 (make-instance 'service
                :name 'change-date
                :desc "Change the date of the mission."
                :input-set `(,(make-parameterf :name 'new-date
                                               :values 'date))
                :output-set '()
                :pre '()
                :post
                (make-postf :atts `(,(make-attr-val
                                      :name 'date
                                      :value 'new-date))) ))

(change-time
 (make-instance 'service
                :name 'change-time
                :desc "Change the time of the mission."
                :input-set `(,(make-parameterf :name 'new-time
                                               :values 'time))
                :output-set '()
                :pre '()
                :post
                (make-postf :atts `(,(make-attr-val
                                      :name 'time
                                      :value 'new-time))) ))

(change-ac-info
 (make-instance 'service
                :name 'change-ac-info
                :desc "Change the information on the aircraft, their times,
            and their configuration."
                :input-set `(,(make-parameterf :name 'new-ac-info
                                               :values 'ac-info))
                :output-set '()
                :pre '()
                :post
                (make-postf :atts `(,(make-attr-val
                                      :name 'ac-info
```

```
                                        :value 'new-ac-info))) ))

      (change-aircrew-list
       (make-instance 'service
                      :name 'change-aircrew-list
                      :desc "Change the list of aircrew."
                      :input-set `(,(make-parameterf :name 'new-aircrew-list
                                                     :values 'aircrew-list))
                      :output-set '()
                      :pre '()
                      :post
                      (make-postf :atts `(,(make-attr-val
                                            :name 'aircrew-list
                                            :value 'new-aircrew-list))) ))

      (change-status
       (make-instance 'service
                      :name 'change-status
                      :desc "Change the status of the mission."
                      :input-set `(,(make-parameterf :name 'new-status
                                                     :values 'status))
                      :output-set '()
                      :pre '()
                      :post
                      (make-postf :atts `(,(make-attr-val
                                            :name 'status
                                            :value 'new-status))) )) )

   (make-instance 'generic-class
                  :name 'mission
                  :desc "The class represents one mission. The information
kept on a mission are the aircraft, aircrew, time, and range information."
                  :state-space (list date mission-type ac-info
                                     aircrew-list time
                                     range-info status)
                  :services (list get-aircraft get-duration
                                  get-aircrew change-date
                                  get-date get-config all-aircrew
                                  get-mission-type get-range-info
                                  change-time change-ac-info
                                  change-aircrew-list
                                  change-status)
                  :inheritance '()
                  :whole-part '()
                  :relation `(,(make-relation :name 'uses/used-by
                                              :class1 'plans-and-scheduling
                                              :range1 '(0 n)
                                              :class2 'mission
                                              :range2 '(1 1)))
                  :verif '()) ))

(setf plans-and-scheduling
  (let*
    ((range
       (make-instance 'attribute
                      :name 'range
```

```
                    :desc "The range schedule."
                    :a-set (make-attrs :base 'class
                                       :lower 'range-schedule) ))

(missions
 (make-instance 'attribute
                    :name 'missions
                    :desc "The missions that have been scheduled."
                    :a-set (make-attrs :base `(,(make-attrs :base 'class
                                                            :lower 'mission))) ))

(mission-request
 (make-instance 'service
                    :name 'mission-request
                    :desc "A request for the scheduling of a mission."
                    :input-set `(,(make-parameterf :name 'ac-list
                                                   :values '((:c aircraft)
                                                             (:a aircraft configuration)))
                                 ,(make-parameterf :name 'list-of-aircrew
                                                   :values '((:c aircrew)))
                                 ,(make-parameterf :name 'duration
                                                   :values '(:a schedule-event duration))
                                 ,(make-parameterf :name 'range-info
                                                   :values '(:a mission range-info)))
                    :output-set '()
                    :pre '()
                    :post
                    (make-postf :atts `(,(make-attr-val
                                          :name 'missions
                                          :value '(cons new-mission missions)))
                                :messages '((aircrew get-sched)
                                            (aircraft get-sched)
                                            (mission create)
                                            (aircraft-schedule add-mission)
                                            (aircrew-schedule add-mission)
                                            (range-schedule add-mission))) ))

(mission-complete
 (make-instance 'service
                    :name 'mission-complete
                    :desc "A mission has been completed."
                    :input-set `(,(make-parameterf :name 'the-mission
                                                   :values '(:c mission))
                                 ,(make-parameterf :name 'hours
                                                   :values '(:a mission ac-info))
                                 ,(make-parameterf :name 'crew
                                                   :values '(:a mission aircrew-list))
                                 ,(make-parameterf :name 'date
                                                   :values 'int)
                                 ,(make-parameterf :name 'time
                                                   :values '(:a mission time)))
                    :output-set '()
                    :pre '(member mission missions)
                    :post
                    (make-postf :messages '((aircraft get-config)
                                            (aircraft-part update-flight-hours)
                                          (aircrew update-hours)
                                          (mission change-date)
                                          (mission change-time)
                                          (mission change-aircrew-list)
                                          (mission change-ac-info)
                                          (mission change-status))) ))

      (cancel-mission
       (make-instance 'service
                      :name 'cancel-mission
                      :desc "A missic.ı is canceled."
                      :input-set `(,(make-parameterf :name 'the-mission
                                                     :values '(:c mission)))
                      :output-set '()
                      :pre '(member the-mission missions)
                      :post
                      (make-postf :messages '((aircraft-schedule remove-mission)
                                              (mission get-date)
                                              (mission get-duration)
                                              (mission get-config)
                                              (aircrew-schedule remove-mission)
                                              (mission get-mission-type)
                                              (range-schedule remove-mission)
                                              (mission get-aircraft)
                                              (mission get-range-info)
                                              (mission all-aircrew)
                                              (aircraft get-sched)
                                              (aircrew get-sched)
                                              (mission change-status))) )) )

   (make-instance 'generic-class
                  :name 'plans-and-scheduling
                  :desc "The class represents the plans and scheduling shop,
 which schedules missions. The class accepts mission requests, and schedules
 the range, aircraft, and aircrew for a mission. It cancels missions and releases
 the resources assigned to the mission. It also updates mission information and
 aircrew and aircraft hours when a mission is complete."
                  :state-space (list range missions)
                  :services (list mission-request mission-complete
                                  cancel-mission)
                  :inheritance '()
                  :whole-part '()
                  :relation `(,(make-relation :name 'uses/used-by
                                              :class1 'plans-and-scheduling
                                              :range1 '(0 n)
                                              :class2 'mission
                                              :range2 '(1 1)))
                  :verif '()) ))

 (setf schedule-event
   (let*
     ((day
        (make-instance 'attribute
                       :name 'day
                       :desc "The day the event is scheduled for."
                       :a-set (make-attrs :base 'int) ))
        (start-time
         (make-instance 'attribute
                        :name 'start-time
                        :desc "The start time of the event."
                        :a-set (make-attrs :base 'real) ))

        (duration
         (make-instance 'attribute
                        :name 'duration
                        :desc "The duration of the event."
                        :a-set (make-attrs :base 'real) ))

        (get-day
         (make-instance 'service
                        :name 'get-day
                        :desc "Return the day of the event."
                        :input-set '()
                        :output-set `(,(make-parameterf :values 'day))
                        :pre '()
                        :post '() ))

        (get-start
         (make-instance 'service
                        :name 'get-start
                        :desc "Return the start time of the event."
                        :input-set '()
                        :output-set `(,(make-parameterf :values 'start-time))
                        :pre '()
                        :post '() ))

        (get-duration
         (make-instance 'service
                        :name 'get-duration
                        :desc "Return the duration of the event."
                        :input-set '()
                        :output-set `(,(make-parameterf :values 'duration))
                        :pre '()
                        :post '() )) )

    (make-instance 'generic-class
                   :name 'schedule-event
                   :desc "The class represents an event on a schedule.  It keeps
information on the day, time, and duration of an event."
                   :state-space (list day start-time duration)
                   :services (list get-day get-start get-duration)
                   :inheritance '()
                   :whole-part '()
                   :relation '()
                   :verif '() ) ))

(setf aircrew-schedule-event
  (let*
    ((type-of-mission
        (make-instance 'attribute
                       :name 'type-of-mission
```

```
                        :desc "The type of the mission"
                        :a-set (make-attrs :base 'enum
                                           :lower '(test eval)) ))

          (the-aircraft
           (make-instance 'attribute
                        :name 'the-aircraft
                        :desc "The aircraft flown in."
                        :a-set (make-attrs :base 'class
                                           :lower 'aircraft) ))

          (get-type
           (make-instance 'service
                        :name 'get-type
                        :desc "Return the mission type."
                        :input-set '()
                        :output-set `(,(make-parameterf :values 'type-of-mission))
                        :pre '()
                        :post '() ))

          (get-aircraft
           (make-instance 'service
                        `:name 'get-aircraft
                        :desc "Return the aircraft flown in."
                        :input-set '()
                        :output-set `(,(make-parameterf :values 'the-aircraft))
                        :pre '()
                        :post '() )) )

    (make-instance 'generic-class
                  :name 'aircrew-schedule-event
                  :desc "The class represents one aircraft ride for an aircrew.
It maintains information on the mission type, and the aircraft flown in."
                  :state-space (list type-of-mission the-aircraft)
                  :services (list get-type get-aircraft)
                  :inheritance '(schedule-event)
                  :whole-part `(,(make-relation :class1 'aircrew-schedule
                                                :range1 '(0 n)
                                                :class2 'aircrew-schedule-event
                                                :range2 '(1 1)))
                  :relation '()
                  :verif '() ) ))

(setf aircraft-schedule-event
  (let*
    ((configuration
          (make-instance 'attribute
                        :name 'configuration
                        :desc "The configuration of the aircraft."
                        :a-set (make-attrs :base 'attrib
                                           :lower 'aircraft
                                           :upper 'configuration) ))

          (get-config
           (make-instance 'service
                        :name 'get-config
```

```
                    :desc "Return the aircraft configuration."
                    :input-set '()
                    :output-set `(,(make-parameterf :values 'configuration))
                    :pre '()
                    :post '() )) )

   (make-instance 'generic-class
                 :name 'aircraft-schedule-event
                 :desc "The class represents one scheduled flight for an aircraft.
It maintains information on the configuration of the aircraft for the flight."
                 :state-space (list configuration)
                 :services (list get-config)
                 :inheritance '(schedule-event)
                 :whole-part `(,(make-relation :class1 'aircraft-schedule
                                               :range1 '(0 n)
                                               :class2 'aircraft-schedule-event
                                               :range2 '(1 1)))
                 :relation '()
                 :verif '()) ))

(setf range-schedule-event
  (let*
    ((ac
      (make-instance 'attribute
                    :name 'ac
                    :desc "A list of aircraft."
                    :a-set (make-attrs :base `(,(make-attrs :base 'class
                                                            :lower 'aircraft))) ))

     (range-use
      (make-instance 'attribute
                    :name 'range-use
                    :desc "The altitudes, airspace and facilities needed."
                    :a-set (make-attrs :base 'attrib
                                       :lower 'mission
                                       :upper 'range-info) ))

     (get-aircraft
      (make-instance 'service
                    :name 'get-aircraft
                    :desc "Return the aircraft in the mission."
                    :input-set '()
                    :output-set `(,(make-parameterf :values 'ac))
                    :pre '()
                    :post '() ))

     (get-range-info
      (make-instance 'service
                    :name 'get-range-info
                    :desc "Return range use information."
                    :input-set '()
                    :output-set `(,(make-parameterf :values 'range-use))
                    :pre '()
                    :post '() )))

   (make-instance 'generic-class
                    :name 'range-schedule-event
                    :desc "The class represents one mission scheduled on the range.
It maintains information on the aircraft involved in the mission and the altitudes,
airspace, and range facilities needed."
                    :state-space (list ac range-use)
                    :services (list get-aircraft get-range-info)
                    :inheritance '(schedule-event)
                    :whole-part `(,(make-relation :class1 'range-schedule
                                                   :range1 '(0 n)
                                                   :class2 'range-schedule-event
                                                   :range2 '(1 1)))
                    :relation '()
                    :verif '()) ))

(setf aircrew-schedule
  (let*
    ((schedule
        (make-instance 'attribute
                          :name 'schedule
                          :desc "A schedule for an aircrew."
                          :a-set (make-attrs :base `(,(make-attrs :base 'class
                                                                   :lower 'aircrew-schedule-
event))) ))

     (the-aircrew
        (make-instance 'attribute
                          :name 'the-aircrew
                          :desc "The aircrew member whose schedule this is."
                          :a-set (make-attrs :base 'class
                                             :lower 'aircrew) ))

     (add-mission
        (make-instance 'service
                          :name 'add-mission
                          :desc "Add a mission to the schedule."
                          :input-set `(,(make-parameterf :name 'day
                                                         :values '(:a aircrew-schedule-event day))
                                       ,(make-parameterf :name 'start-time
                                                         :values '(:a aircrew-schedule-event start-
time))
                                       ,(make-parameterf :name 'duration
                                                         :values '(:a aircrew-schedule-event
duration))
                                       ,(make-parameterf :name 'an-aircraft
                                                         :values '(:c aircraft))
                                       ,(make-parameterf :name 'mission-type
                                                         :values
                                                         '(:a aircrew-schedule-event type-of-
mission)))
                          :output-set '()
                          :pre '()
                          :post
                          (make-postf :atts `(,(make-attr-val
                                                 :name 'schedule
                                                 :value '(cons new-mission schedule)))
                                      :messages '((aircrew-schedule-event create))) ))
```

```
          (remove-mission
           (make-instance 'service
                          :name 'remove-mission
                          :desc "Remove a mission from the schedule."
                          :input-set `(,(make-parameterf :name 'day
                                                         :values '(:a aircrew-schedule-event day))
                                       ,(make-parameterf :name 'start-time
                                                         :values '(:a aircrew-schedule-event start-
time))
                                       ,(make-parameterf :name 'duration
                                                         :values '(:a aircrew-schedule-event
duration))
                                       ,(make-parameterf :name 'an-aircraft
                                                         :values '(:c aircraft))
                                       ,(make-parameterf :name 'mission-type
                                                         :values
                                                         '(:a aircrew-schedule-event type-of-
mission)))
                          :output-set '()
                          :pre '()
                          :post
                          (make-postf :atts `(,(make-attr-val
                                                :name 'schedule
                                                :value '(delete old-mission schedule)))
                                      :messages '((aircrew-schedule-event get-day)
                                                  (aircrew-schedule-event get-start)
                                                  (aircrew-schedule-event get-duration)
                                                  (aircrew-schedule-event get-aircraft)
                                                  (aircrew-schedule-event delete)
                                                  (aircrew-schedule-event get-type))) ))

          (get-sched
           (make-instance 'service
                          :name 'get-sched
                          :desc "Return the schedule."
                          :input-set '()
                          :output-set `(,(make-parameterf :values 'schedule))
                          :pre '()
                          :post '() )) )

     (make-instance 'generic-class
                    :name 'aircrew-schedule
                    :desc "The class represents one mission schedule for an aircrew.
It maintains information on the aircrew member, and the schedule for the member."
                    :state-space (list schedule the-aircrew)
                    :services (list add-mission remove-mission
                                    get-sched)
                    :inheritance `()
                    :whole-part `(,(make-relation :class1 'aircrew-schedule
                                                  :range1 '(0 n)
                                                  :class2 'aircrew-schedule-event
                                                  :range2 '(1 1)))
                    :relation `(,(make-relation :name 'has/of
                                                :class1 'aircrew
                                                :range1 '(1 1)
```

```
                                                          :class2 'aircrew-schedule
                                                          :range2 '(1 1)))
                     :verif '()) ))

(setf aircraft-schedule
  (let*
    ((schedule
        (make-instance 'attribute
                             :name 'schedule
                             :desc "The flight schedule for an aircraft."
                             :a-set (make-attrs :base `(,(make-attrs :base 'class
                                                                     :lower 'aircraft-schedule-
event))) ))

       (the-aircraft
        (make-instance 'attribute
                             :name 'the-aircraft
                             :desc "The aircraft the schedule is for."
                             :a-set (make-attrs :base 'class
                                                :lower 'aircraft) ))

       (add-mission
        (make-instance 'service
                             :name 'add-mission
                             :desc "Add a mission to the schedule."
                             :input-set `(,(make-parameterf :name 'day
                                                            :values '(:a aircraft-schedule-event day))
                                          ,(make-parameterf :name 'start-time
                                                            :values '(:a aircraft-schedule-event start-
time))
                                          ,(make-parameterf :name 'duration
                                                            :values '(:a aircraft-schedule-event
duration))
                                          ,(make-parameterf :name 'config
                                                            :values '(:a aircraft configuration)))
                             :output-set '()
                             :pre '()
                             :post
                             (make-postf :atts `(,(make-attr-val
                                                      :name 'schedule
                                                      :value '(cons new-mission schedule)))
                                         :messages '((aircraft-schedule-event create))) ))

       (remove-mission
        (make-instance 'service
                             :name 'remove-mission
                             :desc "Remove a mission from the schedule."
                             :input-set `(,(make-parameterf :name 'day
                                                            :values '(:a aircraft-schedule-event day))
                                          ,(make-parameterf :name 'start-time
                                                            :values '(:a aircraft-schedule-event start-
time))
                                          ,(make-parameterf :name 'duration
                                                            :values '(:a aircraft-schedule-event
duration))
                                          ,(make-parameterf :name 'config
                                                     :values '(:a aircraft configuration)))
                         :output-set '()
                         :pre '()
                         :post
                         (make-postf :atts `(,(make-attr-val
                                                      :name 'schedule
                                                      :value '(delete mission schedule)))
                                    :messages '((aircraft-schedule-event get-day)
                                                         (aircraft-schedule-event get-start)
                                                         (aircraft-schedule-event get-duration)
                                                         (aircraft-schedule-event get-config)
                                                         (aircraft-schedule-event delete))) ))

        (get-sched
         (make-instance 'service
                         :name 'get-sched
                         :desc "Return the aircraft fligh schedule."
                         :input-set '()
                         :output-set `(,(make-parameterf :values 'schedule))
                         :pre '()
                         :post '() )) )

   (make-instance 'generic-class
                   :name 'aircraft-schedule
                   :desc "The class represents the schedule for an aircraft.
The aircraft name and the schedule are maintained."
                   :state-space (list schedule the-aircraft)
                   :services (list add-mission remove-mission get-sched)
                   :inheritance '()
                   :whole-part `(,(make-relation :class1 'aircraft-schedule
                                                  :range1 '(0 n)
                                                  :class2 'aircraft-schedule-event
                                                  :range2 '(1 1)))
                   :relation `(,(make-relation :name 'has-a/for-a
                                                 :class1 'aircraft
                                                 :range1 '(1 1)
                                                 :class2 'aircraft-schedule
                                                 :range2 '(1 1)))
                   :verif '()) ))

(setf range-schedule
  (let*
    ((schedule
         (make-instance 'attribute
                         :name 'schedule
                         :desc "The schedule for the range."
                         :a-set (make-attrs :base `(,(make-attrs :base 'class
                                                                  :lower 'range-schedule-event)))
))

        (add-mission
         (make-instance 'service
                         :name 'add-mission
                         :desc "Add a mission to the schedule."
                         :input-set `(,(make-parameterf :name 'day
                                                     :values '(:a range-schedule-event day))
```

```
                                   ,(make-parameterf :name 'time
                                                     :values '(:a mission time))
                                   ,(make-parameterf :name 'aircraft-list
                                                     :values '(:a range-schedule-event ac))
                                   ,(make-parameterf :name 'range-info
                                                     :values '(:a range-schedule-event range-
use)))
                            :output-set '()
                            :pre '()
                            :post
                            (make-postf :atts `(,(make-attr-val
                                                  :name 'schedule
                                                  :value '(cons new-mission schedule)))
                                  :messages '((range-schedule-event create))) ))

       (remove-mission
        (make-instance 'service
                            :name 'remove-mission
                            :desc "Removᵉ a mission from the schedule."
                            :input-set `(,(make-parameterf :name 'day
                                                     :values '(:a range-schedule-event day))
                                   ,(make-parameterf :name 'time
                                                     :values '(:a mission time))
                                   ,(make-parameterf :name 'aircraft-list
                                                     :values '(:a range-schedule-event ac))
                                   ,(make-parameterf :name 'range-info
                                                     :values '(:a range-schedule-event range-
use)))
                            :output-set '()
                            :pre '()
                            :post
                            (make-postf :atts `(,(make-attr-val
                                                  :name 'schedule
                                                  :value '(delete mission schedule)))
                                  :messages '((range-schedule-event get-day)
                                              (range-schedule-event get-start)
                                              (range-schedule-event get-duration)
                                              (range-schedule-event get-aircraft)
                                              (range-schedule-event get-range-info)
                                              (range-schedule-event delete))) ))

       (get-sched
        (make-instance 'service
                            :name 'get-sched
                            :desc "Return the range schedule."
                            :input-set '()
                            :output-set `(,(make-parameterf :values 'schedule))
                            :pre '()
                            :post '() )) )

    (make-instance 'generic-class
                        :name 'range-schedule
                        :desc "The class represents the schedule for the range."
                        :state-space (list schedule)
                        :services (list add-mission remove-mission get-sched)
                        :inheritance '()
```

```
                    :whole-part `(,(make-relation :class1 'range-schedule
                                                  :range1 '(0 n)
                                                  :class2 'range-schedule-event
                                                  :range2 '(1 1)))
                    :relation `()
                    :verif '() ) ))

(setf aircraft-parking
  (let*
    ((spots
      (make-instance 'attribute
                     :name 'spots
                     :desc "A list of flight line spots for parking aircraft."
                     :a-set (make-attrs :base `(,(make-attrs :base 'class
                                                             :lower 'flight-line-spots))) ))

     (hangars
      (make-instance 'attribute
                     :name 'hangars
                     :desc "A list of hangar spots for aircraft maintenance."
                     :a-set (make-attrs :base `(,(make-attrs :base 'class
                                                             :lower 'hangar))) ))

     (type
      (make-instance 'attribute
                     :name 'type
                     :desc "The type of maintenance of the aircraft."
                     :a-set (make-attrs :base 'attrib
                                        :lower 'maintenance-history
                                        :upper 'type) ))

     (requests-pending
      (make-instance 'attribute
                     :name 'requests-pending
                     :desc "Requests for hangar slots that are pending."
                     :a-set (make-attrs :base `(,(make-attrs :base 'class
                                                             :lower 'support-shop)
                                                ,(make-attrs :base 'class
                                                             :lower 'aircraft-part)
                                                ,(make-attrs :base 'attrib
                                                             :lower 'maintenance-history
                                                             :upper 'type))) ))

     (add-hangar
      (make-instance 'service
                     :name 'add-hangar
                     :desc "Add a hangar spot."
                     :input-set `(,(make-parameterf :name 'new-hangar
                                                    :values '(:c hangar)))
                     :output-set '()
                     :pre '(not (member new-hangar hangars))
                     :post
                     (make-postf :atts `(,(make-attr-val
                                           :name 'hangars
                                           :value '(cons new-hangar hangars)))
                                 :messages '((hangar create))) ))
```

```
(remove-hangar
 (make-instance 'service
                :name 'remove-hangar
                :desc "Remove a hangar spot."
                :input-set `(,(make-parameterf :name 'old-hangar
                                               :values '(:c hangar)))
                :output-set '()
                :pre '(member old-hangar hangars)
                :post
                (make-postf :atts `(,(make-attr-val
                                      :name 'hangars
                                      :value '(delete old-hangar hangars)))
                            :messages '((hangar delete))) ))

(release-hangar
 (make-instance 'service
                :name 'release-hangar
                :desc "Release the use of a hangar spot by an aircraft."
                :input-set `(,(make-parameterf :name 'the-hangar
                                               :values '(:c hangar)))
                :output-set `(,(make-parameterf :values '(:c flight-line-spots)))
                :pre '()
                :post
                (make-postf :atts `(,(make-attr-val
                                      :name 'requests-pending
                                      :value '(last requests-pending)))
                            :messages '((hangar release)
                                        (support-shop hangar-available)
                                        (hangar new-aircraft)
                                        (aircraft-part get-aircraft)
                                        (flight-line-spots occupied)
                                        (flight-line-spots fill))) ))

(schedule-hangar
 (make-instance 'service
                :name 'schedule-hangar
                :desc "Schedule the use of a hangar spot"
                :input-set `(,(make-parameterf :name 'ss
                                               :values '(:c support-shop))
                             ,(make-parameterf :name 'ap
                                               :values '(:c aircraft-part))
                             ,(make-parameterf :name 'a-type
                                               :values '(:a maintenance-history type)))
                :output-set `()
                :pre '()
                :post
                (make-postf :atts `(,(make-attr-val
                                      :name 'requests-pending
                                      :value '(append requests-pending
                                                      '(ss ap a-type))))
                            :messages '((hangar available)
                                        (hangar new-aircraft)
                                        (support-shop hangar-available)
                                        (aircraft-part get-aircraft)
                                        (flight-line-spots occupied)
```

```
                                                (flight-line-spots empty))) ))

          (add-spot
           (make-instance 'service
                          :name 'add-spot
                          :desc "Add a flight line spot to the list of all spots."
                          :input-set `(,(make-parameterf :name 'new-spot
                                                         :values '(:c flight-line-spots)))
                          :output-set '()
                          :pre '(not (member new-spot spots))
                          :post
                          (make-postf :atts `(,(make-attr-val :name 'spots
                                                              :value '(cons new-spot spots)))
                                      :messages '((flight-line-spots create))) ))

          (delete-spot
           (make-instance 'service
                          :name 'delete-spot
                          :desc "delete a spot from the list of all spots."
                          :input-set `(,(make-parameterf :name 'old-spot
                                                         :values '(:c flight-line-spots)))
                          :output-set '()
                          :pre '(member old-spot spots)
                          :post
                          (make-postf :atts `(,(make-attr-val :name 'spots
                                                              :value '(delete old-spot spots)))
                                      :messages '((flight-line-spots delete))) )) )

      (make-instance 'generic-class
                     :name 'aircraft-parking
                     :desc "The class represents all aircraft parking in hangars
  and flight line spots.  A list of hangars and flight line spots is mainained.  The
  class schedules te use of the hangars and flight line spots."
                     :state-space (list spots hangars type requests-pending)
                     :services (list add-hangar release-hangar schedule-hangar
                                     add-spot delete-spot remove-hangar)
                     :inheritance '()
                     :whole-part `(,(make-relation :class1 'squadron
                                                   :range1 '(1 n)
                                                   :class2 'aircraft-parking
                                                   :range2 '(1 1))
                                   ,(make-relation :class1 'aircraft-parking
                                                   :range1 '(1 n)
                                                   :class2 'hangar
                                                   :range2 '(1 1))
                                   ,(make-relation :class1 'aircraft-parking
                                                   :range1 '(1 n)
                                                   :class2 'flight-line-spots
                                                   :range2 '(1 1)))
                     :relation '()
                     :verif '() ) ))

  (setf hangar
    (let*
      ((occupied-by
         (make-instance 'attribute
                          :name 'occupied-by
                          :desc "The aircraft in the hangar slot."
                          :a-set (make-attrs :base 'class
                                             :lower 'aircraft) ))

      (new-aircraft
       (make-instance 'service
                          :name 'new-aircraft
                          :desc "Add an aircraft to the hangar slot."
                          :input-set `(,(make-parameterf :name 'ac
                                                  :values '(:c aircraft)))
                          :output-set '()
                          :pre '()
                          :post
                          (make-postf :atts `(,(make-attr-val :name 'occupied-by
                                                      :value 'ac))) ))

      (available
       (make-instance 'service
                          :name 'available
                          :desc "Return true if the hangar slot is available."
                          :input-set '()
                          :output-set `(,(make-parameterf :values 'bool))
                          :pre '()
                          :post '() ))

      (release
       (make-instance 'service
                          :name 'release
                          :desc "Make the hangar available."
                          :input-set '()
                          :output-set `(,(make-parameterf :values 'occupied-by))
                          :pre '()
                          :post
                          (make-postf :atts `(,(make-attr-val :name 'occupied-by
                                                      :value '()))) )) )
   (make-instance 'generic-class
                    :name 'hangar
                    :desc "The class represents one hangar slot for aircraft maintenance.
Whether the hangar is occupied and by what aircraft are maintained."
                    :state-space (list occupied-by)
                    :services (list new-aircraft available release)
                    :inheritance '()
                    :whole-part `(,(make-relation :class1 'aircraft-parking
                                             :range1 '(1 n)
                                             :class2 'hangar
                                             :range2 '(1 1)))
                    :relation '()
                    :verif '()) ))

(setf flight-line-spots
  (let*
    ((the-aircraft
       (make-instance 'attribute
                          :name 'the-aircraft
                          :desc "The aircraft in the spot."
```

```
                           :a-set (make-attrs :base 'class
                                              :lower 'aircraft) ))

      (occupied
       (make-instance 'service
                           :name 'occupied
                           :desc "Return an aircraft if the slot is occupied,
                     otherwise returns false."
                           :input-set '()
                           :output-set `(,(make-parameterf :values 'the-aircraft))
                           :pre '()
                           :post '() ))

      (fill
       (make-instance 'service
                           :name 'fill
                           :desc "Fill the spot."
                           :input-set `(,(make-parameterf :name 'ac
                                                          :values '(:c aircraft)))
                           :output-set '()
                           :pre '(null the-aircraft)
                           :post
                           (make-postf :atts `(,(make-attr-val :name 'the-aircraft
                                                               :value 'ac))) ))

      (empty
       (make-instance 'service
                           :name 'empty
                           :desc "Empty the spot."
                           :input-set '()
                           :output-set `(,(make-parameterf :values 'the-aircraft))
                           :pre '()
                           :post
                           (make-postf :atts `(,(make-attr-val :name 'the-aircraft
                                                               :value '()))) )) )

   (make-instance 'generic-class
                        :name 'flight-line-spots
                        :desc "The class represents one flight line spot for parking
aircraft.  Whether the slot is occupied and by what aircraft is maintained."
                        :state-space (list the-aircraft)
                        :services (list occupied fill empty)
                        :inheritance '()
                        :whole-part `(,(make-relation :class1 'aircraft-parking
                                                      :range1 '(1 n)
                                                      :class2 'flight-line-spots
                                                      :range2 '(1 1)))
                        :relation '()
                        :verif '()) ))

(defparameter *list-of-classes* (list squadron flight aircraft aircraft-part periodic-task
                                   periodic-maintenance repair-symptoms
                                   maintenance-history people support-person
                                   support-shop aircrew mission plans-and-scheduling
                                   schedule-event aircrew-schedule-event
                                   aircraft-schedule-event range-schedule-event
```

```
                                        aircrew-schedule aircraft-schedule range-schedule
                                        aircraft-parking hangar flight-line-spots))

;; Necessary classes are those classes that cannot be deleted.  Also associated
;; with each classes are two lists, possibly null, of the attributes and services
;; that cannot be deleted.  The first list are the attributes and the second the
;; services.  The names and components of the necessary classes, attributes and
;; services can change.
(defparameter *necessary-classes*
  '((aircraft (tail-number) ())
    (aircrew () (get-sched))
    (aircraft-part () ())))
```

OAKSNO.LISP

Model Evaluation Code

```
(in-package 'oaks)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; OAKS non-domain specific model evaluation
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Model Evaluation
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;------------------------------------------------------------------------
;; Evaluate the entire model to ensure all class names are unique.
;;------------------------------------------------------------------------
(defun unique-class-names ()
 (check-unique (first (class-namel)) (rest (class-namel))))

(defun check-unique (name l)
 (symbolp name)
 (listp l)
 (cond ((null l) '())
         ((member name l)
          (cons name (check-unique (first l) (rest l))))
         (t (check-unique (first l) (rest l))) ))

;;------------------------------------------------------------------------
;; Evaluate the entire model for missing classes that are part of attributes.
;;------------------------------------------------------------------------
(defun model-att-class-check ()
 (remove-null (mapca, #'missing-att-classes *list-of-classes*)))

;;------------------------------------------------------------------------
;; Evaluate the entire model for missing attributes of other classes that
;; are part of attributes.
;;------------------------------------------------------------------------
(defun model-att-att-check ()
 (remove-null (mapcar #'missing-att-atts *list-of-classes*)))

;;------------------------------------------------------------------------
;; Evaluate the entire model for classes, services, and attributes used in
;; services that are not part of the model.
;;------------------------------------------------------------------------
(defun model-serv-att-class-check ()
 (remove-null (mapcar #'missing-serv-att-class *list-of-classes*)))

;;------------------------------------------------------------------------
;; Evaluate the entire model for whole/part structures that are not repeated
;; exactly in their respective classes.
;;------------------------------------------------------------------------
(defun model-wp-check ()
 (remove-null (mapcar #'wp-class-check *list-of-classes*)))

;;------------------------------------------------------------------------
;; Evaluate the entire model for relation structures that are not repeated
;; exactly in their respective classes.
;;------------------------------------------------------------------------
(defun model-rel-check ()
  (remove-null (mapcar #'rel-class-check *list-of-classes*)))

;;-------------------------------------------------------------------
;; Evaluate the entire model to ensure the class names of the parents of
;; classes exist in the model.
;;-------------------------------------------------------------------
(defun model-parent-check ()
  (remove-null (mapcar #'parent-check *list-of-classes*)))


;;-----------------------------------------------------------------------
;; Evaluate the entire model to ensure the names of the input set parameters
;; of services are:
;;   - not the same as another parameter in the same input set
;;   - not the same as an attribute within its class
;;   - not the same as the legal set of attributes
;;-----------------------------------------------------------------------
(defun model-input-set-names ()
  (remove-null (mapcar #'check-serv-input-set *list-of-classes*)))

;;---------------------------------------------------------------------
;; Return a list consisting of the class names
;;--------------------------------------------------------------------
(defun class-namel ()
  (mapcar #'name *list-of-classes*))

;;---------------------------------------------------------------------
;; Check to see if the two class names in each relation in the whole-part
;; and other relation slots of a class are different.  A class cannot be
;; related to itself.
;;---------------------------------------------------------------------
(defun model-relation-classes-different ()
  (remove-null (mapcar #'class-relation-classes-different *list-of-classes*)))

;;----------------------------------------------------------------------
;; Remove repreated relations in a class.  These are relations that are
;; exactly the same.
;;----------------------------------------------------------------------
(defun model-remove-repeated-relations ()
  (mapcar #'class-remove-repeated-relations *list-of-classes*))

;;---------------------------------------------------------------------
;; Remove repeated messages in a class.  These are messages with the
;; same class and service.
;;---------------------------------------------------------------------
(defun model-remove-repeated-messages ()
  (mapcar #'class-remove-repeated-messages *list-of-classes*))

;;---------------------------------------------------------------------
;; Service GR7
;; Trace the message flow through the model given an initial class name
;; and service name.
;;---------------------------------------------------------------------
(defun trace-messages (class-name service-name)
  (symbolp class-name)
  (symbolp service-name)
```

```
 ;; check if the class and service are valid
 (if (null (check-messages (list (list class-name service-name))))
    (trace-message-loop (list (list class-name service-name)))
    (progn
        (princ #\newline)
        (princ "The class and service are not valid.")
        (princ #\newline)
        (values)) ))

(defun trace-message-loop (list-of-pairs)
 (if (null list-of-pairs)
    '()
    (let* ((the-serv (if (or (eql (second (first list-of-pairs)) 'create)
                              (eql (second (first list-of-pairs)) 'delete))
                           '()
                           (return-service (first (first list-of-pairs))
                                           (second (first list-of-pairs)))))
           (new-pairs (cond ((null the-serv) '())
                            ((null (post the-serv)) '())
                            (t
                             (postf-messages (post the-serv))) )))
        (print-message-path (first list-of-pairs) new-pairs)
        (trace-message-loop (append (rest list-of-pairs) new-pairs)) )))

(defun print-message-path (a-pair the-messages)
 (progn
   (princ (first a-pair))
   (princ #\space)
   (princ (second a-pair))
   (princ #\newline)
   (dolist (one-message the-messages)
           (dotimes (i 4)
                    (princ #\space))
           (princ (first one-message))
           (princ #\space)
           (princ (second one-message))
           (princ #\newline))
   (values) ))

;;------------------------------------------------------------------------
;; Evaluate the entire model for classes that are not connected to other classes.
;;------------------------------------------------------------------------
(defun unconnected-classes ()
 (let ((result '()))
   (dolist (one-class *list-of-classes* result)
           (if (not (connectionp one-class))
                (cons (name one-class) result)))))

;;------------------------------------------------------------------------
;; Evaluate all classes in the model to ensure attribute names within the class
;; are unqiue and the names are not equal to the valid sets for attribute
;; values.
;;------------------------------------------------------------------------
(defun model-unique-att-names ()
 (let ((result '()))
   (dolist (one-class *list-of-classes* result)
            (if (not (unique-att-namesp one-class))
                 (setf result (cons (name one-class) result)) ))))

;;-------------------------------------------------------------------------
;; Evaluate all classes in the model to ensure service names within the class
;; are unique.
;;-------------------------------------------------------------------------
(defun model-unique-serv-names ()
 (let ((result '()))
  (dolist (one-class *list-of-classes* result)
            (if (not (unique-serv-namesp one-class))
                 (setf result (cons (name one-class) result)) ))))

;;-------------------------------------------------------------------------
;; Given a new class name, determine if it has any relationship to any other
;; class names in the model. This is determined by comparing the string
;; after the first dash (if any) in the class name to see if it matches
;; a class name already in the model. It assumes the class name does not
;; exactly match any class name in the model.
;;-------------------------------------------------------------------------
(defun class-name-match (new-name)
 (symbolp new-name)
 (let* ((name-string (symbol-name new-name))
         (name-length (length name-string))
         (dash-loc
          (dotimes (i name-length)
                    (if (char= #\- (char name-string i))
                       (return i))
                    (if (eql i (1- name-length))
                       (return '())) )))
  (if (null dash-loc) '()
   (let ((result '()))
    (dolist (one-name (class-namel) result)
                  (if (eql (length (symbol-name one-name)) (- name-length (1+ dash-loc)))
                     (let ((not-match '()))
                      (dotimes (i (length (symbol-name one-name)))
                                (if (not (char= (char (symbol-name one-name) i)
                                                (char name-string (+ i (1+ dash-loc)))))
                                      (setf not-match t)))
                      (if (not not-match)
                            (setf result (cons one-name result))) )))))))

;;-------------------------------------------------------------------------
;; Class GR1
;; Determine if any class's name in the model end in "s".
;;-------------------------------------------------------------------------
(defun model-singular-noun-check ()
 (let ((result '()))
  (dolist (cn (class-namel) result)
            (if (singular-noun-check cn)
                 (setf result (cons cn result))))))

;;-------------------------------------------------------------------------
;; Inheritance GR6
;; Return a list consisting of pairs of class names and nesting level.
;;-------------------------------------------------------------------------
```

```
(defun model-class-depth ()
 (let ((result '()))
  (dolist (c *list-of-classes* result)
        (setf result (cons (list (name c)
                                    (class-depth c)) result)) )))

;;-----------------------------------------------------------------------
;; Inheritance GR9
;; Return all classes that are parents that only have one child class.
;;-----------------------------------------------------------------------
(defun model-two-subclass-check ()
 (remove-null (mapcar #'two-subclass-check *list-of-classes*)))

;;-----------------------------------------------------------------------
;; State Space GR5
;; Return all classes that have other classes with similar attributes.
;;-----------------------------------------------------------------------
(defun model-similar-atts ()
 (let ((result '()))
  (dolist (c *list-of-classes* result)
        (if (not (null (similar-atts c)))
             (setf result (cons (list (name c) (similar-atts c)) result))))))

;;-----------------------------------------------------------------------
;; services GR10
;; Returns the names of classes that do not have message connections with
;; other classes, and do not have children.
;;-----------------------------------------------------------------------
(defun model-message-connectionsp ()
 (let ((result '()))
  (dolist (c *list-of-classes* result)
        (if (not (message-connectionsp c))
             (setf result (cons (name c) result)) ))))

;;-----------------------------------------------------------------------
;; Services GR14
;; Return all classes that have classes with similar services.
;;-----------------------------------------------------------------------
(defun model-similar-servs ()
 (let ((result '()))
  (dolist (c *list-of-classes* result)
        (if (not (null (similar-servs c)))
             (setf result (cons (list (name c) (similar-atts c)) result))))))

;;-----------------------------------------------------------------------
;; Model GR1
;; Returns all classes that have only one attribute.
;;-----------------------------------------------------------------------
(defun model-one-attributep ()
 (let ((result '()))
  (dolist (one-class *list-of-classes* result)
        (if (one-attributep one-class)
             (setf result (cons (name one-class) result)) ))))

;;-----------------------------------------------------------------------
;; Model GR2,10
;; Returns all classes that have only one service.
;;------------------------------------------------------------------------
(defun model-one-servicep ()
 (let ((result '()))
  (dolist (one-class *list-of-classes* result)
          (if (one-servicep one-class)
               (setf result (cons (name one-class) result)) ))))

;;------------------------------------------------------------------------
;; Model GR6
;; Returns a list of class names and their total number of services and attributes.
;;------------------------------------------------------------------------
(defun model-num-att-serv ()
 (mapcar #'num-att-ser *list-of-classes*))

;; Return the average number of attributes and services in a class.
(defun model-ave-att-serv ()
 (round (/ (apply #'+ (mapcar #'second (model-num-att-serv)))
          (length *list-of-classes*))))

;;------------------------------------------------------------------------
;; Model GR7
;; Return any two classes in the model that share 80% or more of their
;; attributes and services.
;;------------------------------------------------------------------------
(defun model-share-att-serv ()
 (remove-null (mapcar #'share-att-serv *list-of-classes*)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Class evaluation
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;------------------------------------------------------------------------
;; Given a class name, return the class.
;;------------------------------------------------------------------------
(defun return-class (a-name)
 (let ((result '()))
  (dolist (one-class *list-of-classes*)
        (if (eql (name one-class) a-name)
           (setf result one-class)))
  result))

;;------------------------------------------------------------------------
;; Return a list of attribute names of a class.
;;------------------------------------------------------------------------
(clos:defgeneric att-names (c)
   (:documentation "Return a list of attribute names given a class.")
   (:method ((c generic-class))
           (mapcar #'name (state-space c))))

;;------------------------------------------------------------------------
;; Return a list of the service names of a class.
;;------------------------------------------------------------------------
(clos:defgeneric ser-names (c)
   (:documentation "Return a list of all the services of a class.")
   (:method ((c generic-class))
               (mapcar #'name (services c)) ))

;;-----------------------------------------------------------------------
;;   Check a class to ensure all classes used in attributes to define the legal
;; set of values are valid classes.
;;   The resulting list will consist of the class name, and lists of pairs
;; of attribute names and missing class names.
;;   (mission (aircrew-list (aircrew aircraft)) (ac-info (aircraft)))
;; where mission is the name of the class, aircrew-list and ac-info are
;; attributes, and aircrew and aircraft are missing class names.
;;-----------------------------------------------------------------------
(clos:defgeneric missing-att-classes (c)
    (:documentation "Check all the attributes of a class to ensure
classes used as a basis for the attributes exist.")
    (:method ((c generic-class))
               (let ((result (remove-null
                                   (mapcar #'atts-classc (state-space c)))))
                   (if (not (null result))
                      (cons (name c) result)
                      '()))))

;;-----------------------------------------------------------------------
;; Check all attributes of a class to ensure all attributes of other classes
;; used to define the legal set of values for an attribute are attributes
;; of the class.
;; This assumes that the class exists, and the name of the class is unique.
;; It checks to see if the attribute of another class used as a basis for an
;; attribute exists in that class.
;;-----------------------------------------------------------------------
(clos:defgeneric missing-att-atts (c)
    (:documentation "Check a class to ensure attributes of other classes
used as a basis for an attribute exist.")
    (:method ((c generic-class))
               (let ((result (remove-null
                                   (mapcar #'atts-attc (state-space c)))))
                   (if (not (null result))
                      (cons (name c) result)
                      '()) )))

;;-----------------------------------------------------------------------
;; Check all services in a class to ensure all classes, services and
;; attributes (both internal and external) used are valid in that they
;; exist in the model.
;;-----------------------------------------------------------------------
(clos:defgeneric missing-serv-att-class (c)
    (:documentation "Check to see if all attributes, classes,
and services exist.")
    (:method ((c generic-class))
               (let ((result '()))
                   (dolist (one-serv (services c))
                            (push (ser-class-att-check one-serv (name c))
                                result))
                   (setf result (remove-null result))
                   (if (not (null result))
                      (cons (name c) result)
                      '() ))))
```

```
;;------------------------------------------------------------------
;; Check to see if all whole/part structure in a class are repeated exactly
;; in the other class of the whole/part structure.  If the whole/part
;; structure does not exist in the other class, 'not-exist is returned.
;; If it does exist, but the structure does not have the same values, the
;; whole/part structure in the other class is returned.
;;------------------------------------------------------------------
(clos:defgeneric wp-class-check (c)
    (:documentation "Check to see if each whole/part structure is
repeated exactly in the other class.")
    (:method ((c generic-class))
                (let ((result '()))
                  (dolist (one-wp (whole-part c))
                          (push (check-wp one-wp (name c)) result))
                  (setf result (remove-null result))
                  (if (not (null result))
                     (cons (name c) result)
                     '()) )))

;;------------------------------------------------------------------
;; Check to see if all other relations in a class are repeated exactly
;; in the other class of the relation structure.  If the structure does
;; not exist in the other class, 'not exist is returned.  If it does
;; exist, but the structure does not have the same slot values, the
;; relation structure in the other class is returned.
;;------------------------------------------------------------------
(clos:defgeneric rel-class-check (c)
    (:documentation "Check to ensure the relation strcutures
are repeated exactly in the other classes of the relations.")
    (:method ((c generic-class))
              (let ((result '()))
                  (dolist (one-r (relation c))
                          (push (check-rel one-r (name c)) result))
                  (setf result (remove-null result))
                  (if (not (null result))
                     (cons (name c) result)
                     '()) )))

;;------------------------------------------------------------------
;; Ensure all classes in the inheritance slot exist in the model.
;;------------------------------------------------------------------
(clos:defgeneric parent-check (c)
    (:documentation "Ensure all classes in the inheritance
slot exist in the model.")
    (:method ((c generic-class))
              (remove-null (mapcar #'parent-exists (inheritance c)) )))

;;------------------------------------------------------------------
;; Check all services in a class to ensure the names of the input set
;; parameters:
;;   - are unique within the input set
;;   - are not the same as attribute names in the class.
;;   - are not the same as the legal set of attribute values.
;;------------------------------------------------------------------
(clos:defgeneric check-serv-input-set (c)
   (:documentation "Check the names of the input set parameters tp ensure
they are unique within the input set and are not the same as local attributes
or legal attribute values.")
   (:method ((c generic-class))
            (let ((result '()))
                (dolist (one-serv (services c))
                        (let ((one-serv-result
                                (check-val-name (input-set one-serv) c)))
                          (if one-serv-result
                             (push (list (name one-serv) one-serv-result)
                                   result))))
                (setf result (remove-null result))
                (if (not (null result))
                   (cons (name c) result)
                   '()) )))

;;-----------------------------------------------------------------------
;; Return a list of the class name and any relations where the the two
;; class names in the relation are the same.  If there are no relations
;; that have the class names the same, return null.
;;-----------------------------------------------------------------------
(clos:defgeneric class-relation-classes-different (c)
   (:documentation "Return the class name and any relations with the
same class name for both classes in the relation, otherwise return null.")
   (:method ((c generic-class))
            (let ((wp-result (remove-null
                               (mapcar #'relation-classes-different
                                       (whole-part c))))
                  (rel-result (remove-null
                               (mapcar #'relation-classes-different
                                       (relation c)))))
                (if (and (null wp-result) (null rel-result))  '()
                  (list (name c) (append wp-result rel-result))))))

;;-----------------------------------------------------------------------
;; Remove repeated relations in the whole-part and relation slots of a class.
;;-----------------------------------------------------------------------
(clos:defgeneric class-remove-repeated-relations (c)
   (:documentation "Remove any repeated relation structures in the
whole-part or relation slots of a class.")
   (:method ((c generic-class))
            (setf (whole-part c)
                  (remove-duplicates (whole-part c) :test #'equalp))
            (setf (relation c)
                  (remove-duplicates (relation c) :test #'equalp))))

;;-----------------------------------------------------------------------
;; Remove repeated messages in the post slot of a class.
;;-----------------------------------------------------------------------
(clos:defgeneric class-remove-repeated-messages (c)
   (:documentation "Remove any repeated messages in the post slot of a class.")
   (:method ((c generic-class))
            (mapcar #'remove-repeated-messages (services c))))

;;-----------------------------------------------------------------------
;; Class GR3.8.9.10.13
;; Determine if a class has no connection to other classes.
;; There are five possible connection:
;; 1. Another class inherits from it.
;; 2. It is part of a whole-part relationship.
;; 3. It is part of other relationships.
;; 4. One of its services is used by another class's services.
;; 5. One of its services calls another class's services.
;;-------------------------------------------------------------------
(clos:defgeneric connectionp (c)
    (:documentation "Check to see if this class has any
connection to other classes in the model.")
    (:method ((c generic-class))
             (or (parentp c)
                  (whole-part-memberp c)
                  (relation-memberp c)
                  (calledp c)
                  (callsp c)) ))

(clos:defgeneric parentp (c)
    (:documentation "Returns true if c is a parent of any other class.")
    (:method ((c generic-class))
             (let ((result '()))
                  (eval (cons 'or
                              (dolist (one-class *list-of-classes*
                                                  result)
                                       (if (member (name c) (inheritance one-class))
                                             (push t result)
                                           (push '() result)) ))))))

(clos:defgeneric whole-part-memberp (c)
    (:documentation "Returns true if there exists a whole-part
relationship in the class.  This assumes the model has been checked
to ensure all whole-part relationships are repeated in the involved classes.")
    (:method ((c generic-class))
             (not (null (whole-part c))) ))

(clos:defgeneric relation-memberp (c)
    (:documentation "Returns true if there exists a relationship
in the class.  Assumes the model has been checked to ensure all relationships
are repeated in the involved classes.")
    (:method ((c generic-class))
             (not (null (relation c))) ))

(clos:defgeneric calledp (c)
    (:documentation "Returns true if any service of the class is called
by a service of another class.")
    (:method ((c generic-class))
             (let ((result '()))
                  (eval (cons 'or
                            (dolist (one-class *list-of-classes* result)
                               (if (services one-class)
                                     (dolist (one-service (services one-class))
                                        (if (post one-service)
                                              (dolist (one-message
                                                         (postf-messages
                                                          (post one-service)))
                                        (push (eql (name c)
                                                (first one-message)) result) ))))))))))

(clos:defgeneric callsp (c)
   (:documentation "Returns true if any service of the class calls the
service of another class.")
   (:method ((c generic-class))
          (let ((result '()))
             (eval (cons 'or
                     (dolist (one-service (services c) result)
                          (if (post one-service)
                             (push (not (null
                                       (postf-messages (post one-service))))
result))))))))

;;-----------------------------------------------------------------------
;; Class GR1
;; Check to see if the class name ends in an "s".
;;-----------------------------------------------------------------------
(defun singular-noun-check (class-name)
 (symbolp class-name)
 (char= #\S
      (char (symbol-name class-name)
          (1- (length (symbol-name class-name))))))

;;-----------------------------------------------------------------------
;; Model GR1
;; Return true if there is only one attribute in the class.
;;-----------------------------------------------------------------------
(clos:defgeneric one-attributep (c)
   (:documentation "Return true if there is less than two attributes in
the class.")
   (:method ((c generic-class))
          (< (length (state-space c)) 2) ))

;;-----------------------------------------------------------------------
;; Model GR2,10
;; Return true if there is less than two services in the class.
;;-----------------------------------------------------------------------
(clos:defgeneric one-servicep (c)
   (:documentation "Return true if there is less than two services in the
class.")
   (:method ((c generic-class))
          (< (length (services c)) 2) ))

;;-----------------------------------------------------------------------
;; Model GR6
;; Return a list consisting of the class name and the total number of
;; attributes and services.
;;-----------------------------------------------------------------------
(clos:defgeneric num-att-ser (c)
   (:documentation "Return a list consisting of teh class name and the
total number of attributes and services.")
   (:method ((c generic-class))
          (list (name c) (+ (length (state-space c))
                          (length (services c)) ))))
```

```
;;-----------------------------------------------------------------------
;; Model GR7
;; Return a class that shares 80% of its attributes and 80% of its
;; services.
;;-----------------------------------------------------------------------
(clos:defgeneric share-att-serv (c)
    (:documentation "Return a class that shares a majority of its
attributes and services.")
    (:method ((c generic-class))
              (let ((ser-match (similar-servs c))
                     (att-match (similar-atts c)))
                  (if (and ser-match att-match)
                     (let ((class-ser-match (mapcar #'first ser-match))
                             (class-att-match (mapcar #'first att-match)))
                      (intersection class-ser-match class-att-match) )))))

;;-----------------------------------------------------------------------
;; Ensure attribute names within the class are unique and they are not equal
;; to the legal set of attribute sets.
;;-----------------------------------------------------------------------
(clos:defgeneric unique-att-namesp (c)
    (:documentation "Ensure attributes names are unique and are not one
of the list of legal attribute sets.")
    (:method ((c generic-class))
              (let ((name-list (att-names c)))
                  (and (not-defined-setp (state-space c))
                     (null (check-unique (first name-list) (rest name-list))) ))))

(defun not-defined-setp (attr-list)
  (listp attr-list)
  (let ((result t))
   (dolist (one-att attr-list result)
             (if (proper-attr-setp (name one-att))
                  (setf result '())))))

;;-----------------------------------------------------------------------
;; Ensure service names within a class are unique.  Return true if they are.
;;-----------------------------------------------------------------------
(clos:defgeneric unique-serv-namesp (c)
    (:documentation "Return true if the service names are unique.")
    (:method ((c generic-class))
          (let ((name-list (ser-names c)))
            (null (check-unique (first name-list) (rest name-list))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Attribute evaluation
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;-----------------------------------------------------------------------
;; Given a class name and an attribute name, return the attribute.
;;-----------------------------------------------------------------------
(defun return-attribute (class-name attribute-name)
  (symbolp class-name)
  (symbolp attribute-name)
  (dolist (one-att (state-space (return-class class-name)))
            (if (eql (name one-att) attribute-name)
               (return one-att) )))

;;---------------------------------------------------------------------------
;; Check one attribute to see if classes used as a basis are valid classes.
;;---------------------------------------------------------------------------
;; Check one attribute to see if there are classes used that are not in the
;; list of classes.  If all the classes are valid, NULL is returned.  If
;; there are invalid classes, a list consisting of the attribute name followed
;; by a list of invalid classes used in that attribute is returned:
;;     (attr-name (class1 ... classn))
(clos:defgeneric atts-classc (a)
    (:documentation "Check one attribute to ensure all classes
 used as the basis for attribute sets are members of the set of classes.")
    (:method ((a attribute))
             (if (listp (attrs-base (a-set a)))
                   (let ((result
                            (remove-null
                             (mapcar #'att-set-classc (attrs-base (a-set a))))))
                     (if (not (null result))
                           (list (name a) result)
                       '()))
                (if (not (null (att-set-classc (a-set a))))
                    (list (name a) (list (att-set-classc (a-set a))))
                  '())) ))

;;  If there is a class name used in the structure, and it is not part
;; of the list of classes, the class name is returned.  If not, NULL
;; is returned.
(defun att-set-classc (ma)
  "Examines one attr structure to ensure any external class used
is a valid class."
  (typep ma 'attrs)
  (if (or (eql (attrs-base ma) 'class) (eql (attrs-base ma) 'attrib))
     (if (not (member (attrs-lower ma) (class-namel)))
           (attrs-lower ma))))

;;------------------------------------------------------------------------
;; Check one attribute to see if attributes of other classes used as a
;; basis are valid attributes of that class.
;;------------------------------------------------------------------------

;; Checks to ensure attributes of other classes used in an attribute
;; exist.  If there are some that do not, the returned list consists
;; of a attribute name, followed by a list consisting of pairs made
;; of the class name and attribute name of the missing attributes:
;;   (attr-name ((class1 att1) ... (classn attn)))
(clos:defgeneric atts-attc (a)
                (:documentation "Check one attribute to ensure attributes of
other classes used as basis for this attribute exist.")
                (:method ((a attribute))
                         (if (listp (attrs-base (a-set a)))
                            (let ((result
                                     (remove-null
                                      (mapcar #'att-set-attc (attrs-base (a-set a))))))
                               (if (not (null result))
                                  (list (name a) result)
                                  '()))
                         (if (not (null (att-set-attc (a-set a))))
                                (list (name a) (att-set-attc (a-set a)))
                                '()) )))

;; Checks one base of an attribute structure to ensure all attributes
;; of other classes used exist in the other classes.  If not, it
;; returns a list consisting of the class name and the attribute name:
;;      (class-name attr-name)
(defun att-set-attc (ma)
 "Examines one attr structure to ensure all external attributes exist."
 (typep ma 'attrs)
 (if (eql (attrs-base ma) 'attrib)
    (if (not (att-exists (attrs-upper ma) (attrs-lower ma)))
          (list (attrs-lower ma) (attrs-upper ma)))))

;;---------------------------------------------------------------------
;; Return True if the attribute exists in the class, or any of the clesses
;; parents.  Allows multiple inheritance.
;;--------------------------------------------------------------------
(defun att-exists (att-name class-name)
 (symbolp att-name)
 (symbolp class-name)
 (cond ((not (member class-name (class-namel)))
         '())
        ((member att-name
                  (att-names (return-class class-name)))
         t)
        ((null (inheritance (return-class class-name)))
         '())
        (t (let ((result '()))
            (eval (cons 'or
                         (dolist
                          (one-class (inheritance (return-class class-name)) result)
                           (push (att-exists att-name one-class) result)))))) ))

;;----------------------------------------------------------------------
;; State Space GR5
;; Compares the attributes of a class with the attributes of other classes.
;; If most, 80%, of the attributes are the same as attributes in other
;; classes, the other class name is returned.
;;----------------------------------------------------------------------
(clos:defgeneric similar-atts (c)
    (:documentation "Returns a list of other classes that have attributes
that are the same as 80% of the attributes of c.")
    (:method ((c generic-class))
              (let ((match-numbers '()))
                  (dolist (one-class *list-of-classes*)
                          (if (not (eql (name one-class) (name c)))
                             (push (have-similar-atts c one-class) match-numbers)))
                  (let ((top-pair (first match-numbers)))
                   (dolist (one-pair match-numbers)
                         (if (> (second one-pair) (second top-pair))
                            (setf top-pair one-pair)))
                   (if (> (second top-pair) (* (length (state-space c)) .8))
                    top-pair
                    '()) ))))

(clos:defgeneric have-similar-atts (c1 c2)
   (:documentation "Returns the number of attributes of C1 match with
attributes of C2.")
   (:method ((c1 generic-class) (c2 generic-class))
          (let ((number-matched 0))
             (dolist (one-att (state-space c1))
                   (if (atts-equal one-att (state-space c2))
                     (setf number-matched (1+ number-matched))))
             (list (name c2) number-matched) )))

(clos:defgeneric atts-equal (one-att ss)
   (:documentation "Returns true if the a-set slot of one-att matches
any a-set slot of ss, which is a list of attributes.")
   (:method ((one-att attribute) (ss list))
          (let ((found-match '()))
             (dolist (att ss found-match)
                   (if (equalp (a-set one-att) (a-set att))
                     (setf found-match t)) ))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Service evaluation
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;-------------------------------------------------------------------------
;; Check each service to ensure all classes, services, and attributes (both
;; external and internal) exist.
;;-------------------------------------------------------------------------
(clos:defgeneric ser-class-att-check (s class-name)
   (:documentation "Check all classes, attributes and services
used in the service to ensure they are valid.")
   (:method ((s service) (class-name symbol))
          (let ((in-set
                   (check-io-set class-name (input-set s)))
                  (out-set
                   (check-io-set class-name (output-set s)))
                  (postc
                   (check-post class-name (post s)))
                  (result '()))
             (if (not (null postc))
                (push (push 'post postc) result))
             (if (not (null out-set))
                (push (push 'output-set out-set) result))
             (if (not (null in-set))
                (push (push 'input-set in-set) result))
             (if (not (null result))
                (push (name s) result)) )))

;; The input and output set can be null or a list of parameterf. Check the values
;; slot of parameterf to ensure all classes and attributes are valid.
;; All external classes have to be members of *list-of-classes*. All
;; external classes must be part of the given class. All other attributes
;; must be part of the class the service is part of.
(defun check-io-set (cn ios)
 "Check the input-set or output-set of a service to ensure all classes and
attributes used are valid."
 (if (null ios) '()
    (remove-null (cons (remove-null (check-parameter cn (first ios)))
                               (check-io-set cn (rest ios)))) ))

(defun check-parameter (cn p)
 "Check one parameter of an input or output set of a service to ensure all
classes and attributes used are valid."
 (typep p 'parameterf)
 (check-para-values cn (parameterf-values p)))

(defun check-para-values (cn val)
 "Checks the value slot of a parameter to ensure validity of
any classes and attributes, both external and internal."
 (cond ((null val) '())
        ((atom val)
         (if (and (not (member val (att-names (return-class cn))))
                  (not (member val '(int real char str bool))))
           (list val)
           '()))
        ((equal (first val) ':c)
         (if (not (member (second val) (class-namel)))
           val))
        ((equal (first val) ':a)
         (if (or (not (member (second val) (class-namel)))
                 (not (att-exists (third val) (second val))))
           val))
        (t (cons (check-para-values cn (first val))
                 (check-para-values cn (rest val)))) ))

;;-----------------------------------------------------------------------
;; Checks the postcondition to determine if the attributes used in the
;; atts slot exist in this class or parents of this class, and determine
;; if the services in the messages slot exist in their proper class.
;;-----------------------------------------------------------------------

;; Return a list consisting of two lists. The first is a list
;; of attributes used in the postconsition attr-val pairs that
;; are not in the class. The second is a list of (class serv)
;; pairs from the messages slot in which either the class does
;; not exist or the service does not exist in that class.
;;    ((att1 ... attn) ((class1 serv1) ... (classn servn)))
(defun check-post (cn p)
 (if (null p) '()
   (remove-null (list (remove-null (check-atts (postf-atts p) cn))
                      (check-messages (postf-messages p))) )))

;; Returns a list consisting of attributes that do not exist in
;; the class : (att1 ... attn)
(defun check-atts (atts-list cn)
 "Returns any attributes used in the name slot of the atts slot
of post that are not in this class."
 (let ((result '()))
   (if (null atts-list)
        '()
```

```
     (dolist (one-att-val atts-list result)
                  (push (check-attr-val one-att-val cn) result)))))

(defun check-attr-val (av cn)
 "Returns the attribute if it is not in the class."
 (typep av 'attr-val)
 (if (not (att-exists (attr-val-name av) cn))
    (attr-val-name av)))

;; Checks the messages to see if the class exists, and if so, if the service
;; exists in that class.  If either is not true, a list consisting of
;; invalid (class service) pairs is returned:
;;   ((class1 serv1) ... (classn servn))
(defun check-messages (m-list)
 "Checks to see if all services in the messages are members of
their class."
 (let ((result '()))
   (if (null m-list)
        '()
     (dolist (one-message m-list result)
                  ;; First check to see if class exists
                  (if (not (member (first one-message) (class-namel)))
                     (push one-message result)
                     (if (not (serv-exists (second one-message) (first one-message)))
                           (push one-message result))) ))))

;;----------------------------------------------------------------------
;; Returns true if the service exists in the class, or any of its parents.
;; Allows multiple inheritance.
;;----------------------------------------------------------------------
(defun serv-exists (serv-name class-name)
 (symbolp serv-name)
 (symbolp class-name)
 (cond ((member serv-name (ser-names (return-class class-name)))
         t)
        ((eql serv-name 'delete)
         t)
        ((eql serv-name 'create)
         t)
        ((null (inheritance (return-class class-name)))
         '())
        (t (let ((result '()))
             (eval (cons 'or
                         (dolist
                          (one-class (inheritance (return-class class-name)) result)
                          (push (serv-exists serv-name one-class) result)))))) ))

;;----------------------------------------------------------------------
;; Returns a service given a class name and a service name.
;;----------------------------------------------------------------------
(defun return-service (class-name service-name)
 (symbolp class-name)
 (symbolp service-name)
 (dolist (one-service (services (return-class class-name)))
          (if (eql (name one-service) service-name)
             (return one-service))))
```

```
;;-------------------------------------------------------------------------
;; Check the input parameter names to ensure they are:
;;  - unique within the input set
;;  - are not the same as any local attribute names
;;  - are not the same as the legal set of attribute values.
;;-------------------------------------------------------------------------
(defun check-val-name (input-set class)
 (listp input-set)
 (if (not (null input-set))
    (append (check-val-name-att input-set class)
            (check-unique-val-name (first input-set) (rest input-set)))))

;; Check to see if the value name is the same as an attribute name or is a
;; member of the legal set of attribute values.
(defun check-val-name-att (input-set class)
 (let ((result '()))
  (dolist (one-para input-set result)
          (if (or (member (parameterf-name one-para) (att-names class))
                  (proper-attr-setp (parameterf-name one-para)))
              (setf result (cons (parameterf-name one-para) result)) ))))

;; Check to see if the value name is unique within the input parameters
(defun check-unique-val-name (one-para para-list)
 (cond ((null para-list) '())
       ((member (parameterf-name one-para) (mapcar #'parameterf-name para-list))
        (cons (parameterf-name one-para)
              (check-unique-val-name (first para-list) (rest para-list))))
       (t (check-unique-val-name (first para-list) (rest para-list))) ))

;;-------------------------------------------------------------------------
;; Remove any repeated messages in the post of a service.
;;-------------------------------------------------------------------------
(clos:defgeneric remove-repeated-messages (s)
    (:documentation "Remove repeated messages in the post of the service.")
    (:method ((s service))
             (if (post s)
                 (if (postf-messages (post s))
                    (setf (postf-messages (post s))
                          (remove-duplicates (postf-messages (post s))
                                             :test #'equal))))))

;;-------------------------------------------------------------------------
;; Service GR2,3,4,5,6,8,9,11,12,13,15,16,17
;; Create templates that create new services that perform the following tasks:
;;  1. Change the value of an attribute to a new, supplied value
;;  2. Return the value of an attribute
;;  3. Add a member to an attribute that is a list.
;;  4. Remove a member from an attribute that is a list.
;;-------------------------------------------------------------------------

;; Creates a service that changes the value of an attribute to a new, given
;; value.  The input parameters is the attribute and any message list.
;; The message list represents any processing that is done by other classes
;; as a result of the change.
;; Before creating the service, check that the attribute exists in the class.
```

```
(clos:defgeneric change-att-template (one-att message-list)
    (:documentation "Create a service that replaces the value of an attribute.")
    (:method ((one-att attribute)(message-list list))
            (let* ((str-name (symbol-name (name one-att)))
                   (new-value (make-symbol (concatenate 'string "new-" str-name))))
               (make-instance 'service
                              :name (make-symbol
                                     (concatenate 'string "change-" str-name))
                              :desc (concatenate 'string "Change the value of "
                                                 str-name)
                              :input-set `(,(make-parameterf
                                             :name new-value
                                             :values (name one-att)))
                              :output-set '()
                              :pre '()
                              :post (make-postf
                                     :atts `(,(make-attr-val :name (name one-att)
                                                             :value new-value))
                                     :messages message-list) ))))

;; Creates a service that returns the value of an attribute.
;; Before creating the service, check the attribute exists in the class.
(clos:defgeneric return-att-template (one-att)
    (:documentation "Create a service that returns the value of an attribute.")
    (:method ((one-att attribute))
            (let ((str-name (symbol-name (name one-att))))
               (make-instance 'service
                              :name (make-symbol
                                     (concatenate 'string "get-" str-name))
                              :desc (concatenate 'string "Return the value of "
                                                 str-name)
                              :input-set '()
                              :output-set `(,(make-parameterf :values (name one-att)))
                              :pre '()
                              :post '() ))))

;; Creates a service that adds an element to an attribute, if that attribute
;; is a list.  The input set is one element that is a component of the attribute.
;; If the components are lists, the input set is one list of the proper elements.
(clos:defgeneric add-element-template (one-att)
    (:documentation "Add an element to an attribute whose value is a list.")
    (:method ((one-att attribute))
            (listp (attrs-base (a-set one-att)))
            (let* ((str-name (symbol-name (name one-att)))
                   (ser-name (make-symbol (concatenate 'string "add-to-" str-name)))
                   (para-name (make-symbol
                               (concatenate 'string "new-" str-name "-element"))))
               (make-instance 'service
                              :name ser-name
                              :desc (concatenate 'string "Add an element to "
                                                 str-name)
                              :input-set `(,(make-parameterf
                                             :name para-name
                                             :values (return-val-list one-att)))
                              :output-set '()
                              :pre `(not (member ,para-name
                                        ,(name one-att)))
                             :post
                             (make-postf :atts `(,(make-attr-val
                                                   :name (name one-att)
                                                   :value `(cons
                                                            ,para-name
                                                            ,(name one-att)) ))) ))))

;; Creates a service that removes an element from an attribute's value, if that
;; attribute is a list.  The input set is one element that is a component of the
;; attribute.  If the components are lists, the input set is one list of the
;; proper elements.
(clos:defgeneric remove-element-template (one-att)
   (:documentation "Remove an element from an attribute whose value is a list.")
   (:method ((one-att attribute))
            (listp (attrs-base (a-set one-att)))
            (let* ((str-name (symbol-name (name one-att)))
                   (ser-name (make-symbol (concatenate 'string "remove-from-"
                                                       str-name)))
                   (para-name (make-symbol
                              (concatenate 'string "old-" str-name "-element"))))
                (make-instance 'service
                             :name ser-name
                             :desc (concatenate 'string "Remove an element from "
                                                str-name)
                             :input-set `(,(make-parameterf
                                           :name para-name
                                           :values (return-val-list one-att)))
                             :output-set '()
                             :pre `(member ,para-name
                                           ,(name one-att))
                             :post
                             (make-postf :atts `(,(make-attr-val
                                                   :name (name one-att)
                                                   :value `(delete
                                                            ,para-name
                                                            ,(name one-att)) ))) ))))


(clos:defgeneric return-val-list (one-att)
   (:documentation "Returns a list of output parameters generated from
the components of the list of one-att")
   (:method ((one-att attribute))
            (let ((result '()))
                (dolist (one-element (attrs-base (a-set one-att)) result)
                    (case (attrs-base one-element)
                        ((class) (setf result (cons (list ':c
                                                          (attrs-lower one-element))
                                                    result)))
                        ((attrib) (setf result (cons (list ':a
                                                           (attrs-lower one-element)
                                                           (attrs-upper one-element))
                                                     result)))
                        (t
                           (setf result (cons (attrs-base one-element) result))) )))))
```

```
;;---------------------------------------------------------------------------
;; Service GR10
;; Return true if the class has message connections with any other class, or
;; if its children have message connections with other classes.
;;--------------------------------------------------------------------------
(clos:defgeneric message-connectionsp (c)
    (:documentation "Return true if the class has message connections with
other classes or has children.")
    (:method ((c generic-class))
              (or (calledp c) (callsp c) (parentp c))))

;;-------------------------------------------------------------------------
;; Services GR14
;; Compares the services of a class with the services of other classes.
;; If most, 80%, of the services are similar to services in other
;; classes, the other class name is returned. The input and output-sets
;; are compared to see if the values of the parameters are the same.
;;-------------------------------------------------------------------------
(clos:defgeneric similar-servs (c)
    (:documentation "Returns a list of other classes that have services
that are the same as 80% of the services of c.")
    (:method ((c generic-class))
              (let ((match-numbers '()))
                   (dolist (one-class *list-of-classes*)
                            (if (not (eql (name one-class) (name c)))
                              (push (have-similar-servs c one-class) match-numbers)))
                   (let ((top-pair (first match-numbers)))
                     (dolist (one-pair match-numbers)
                             (if (> (second one-pair) (second top-pair))
                               (setf top-pair one-pair)))
                     (if (> (second top-pair) (* (length (services c)) .8))
                        top-pair
                        '()) ))))

(clos:defgeneric have-similar-servs (c1 c2)
    (:documentation "Returns the number of services of C1 match with
services of C2.")
    (:method ((c1 generic-class) (c2 generic-class))
              (let ((number-matched 0)
                      ;; the number of services that have not matched yet.
                      ;; prevents one service in c2 matching all the services
                      ;; in c1.
                      (s2-left (services c2)))
                   (dolist (c1-serv (services c1))
                            (dolist (c2-serv s2-left)
                                     (if (serv-match c1-serv (name c1)
                                                     c2-serv (name c2))
                                       (progn
                                        (setf number-matched (1+ number-matched))
                                        (delete c2-serv s2-left)
                                        (return))) ))
                   (list (name c2) number-matched) )))

(clos:defgeneric serv-match (s1 c1n s2 c2n)
    (:documentation "Returns true if the input sets and the outputs sets
of s1 and s2 match.")
```

```
   (:method ((s1 service) (c1n symbol) (s2 service) (c2n symbol))
          (and (para-list-val-match (input-set s1) c1n
                                    (input-set s2) c2n)
               (para-list-val-match (output-set s1) c1n
                                    (output-set s2) c2n)) ))

;; If the length of the two parameter lists are the same, then check each
;; element to see if they are the same.  Each time an element of p1l matches
;; an element of p2l, that element of p2l is removed from values-p2l.  If
;; values p2l is null when all the elements of p1l are checked, then there
;; was a match for each one.
(defun para-list-val-match (p1l cn1 p2l cn2)
 (let* ((values-p1l (values-of-parameters p1l cn1))
        (values-p2l (values-of-parameters p2l cn2)))
  (if (eql (length values-p1l) (length values-p2l))
       (progn
        (dolist (one-p1 values-p1l)
               (dolist (one-p2 values-p2l)
                      (if (equal one-p1 one-p2)
                        (setf values-p2l (delete one-p2 values-p2l)))))
        (if (null values-p2l)
           t
           '()) )
     '()) ))

;; Given a list of parameterf, return a list of the values.  If the value is
;; not a list, it is a local attribute name.  In that case, the a-set slot
;; of the attribute is returned so that the sets can be matched, rather than
;; relying on the naming of the attribute being the same.
(defun values-of-parameters (p-list cn)
 (let ((result '()))
  (dolist (one-p p-list result)
         (cond ((listp (parameterf-values one-p))
                (setf result (cons (parameterf-values one-p) result)))
               ((proper-attr-setp (parameterf-values one-p))
                (setf result (cons (parameterf-values one-p) result)))
               (t
                (setf result (cons
                              (a-set (return-attribute cn (parameterf-values one-p)))
                              result) ))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Inheritance Evaluation
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;-------------------------------------------------------------------------
;; Evalute if the class name exists in the model.  If not, return the class
;; name.
;;-------------------------------------------------------------------------
(defun parent-exists (class-name)
 (symbolp class-name)
 (if (not (member class-name (class-namel)))
    class-name))

;;-------------------------------------------------------------------------
;; Inheritance GR6
;; Determines the depth of a class.  The class sits at level zero.  The parent
;; of a class is at level one.
;;--------------------------------------------------------------------------
(clos:defgeneric class-depth (c)
    (:documentation "Returns the level of nesting for a class.")
    (:method ((c generic-class))
              (if (null (inheritance c)) 0
                   (1+ (apply
                          #'max (mapcar #'class-depth
                                       (mapcar #'return-class (inheritance c))))) )))

;;--------------------------------------------------------------------------
;; Inheritance GR9
;; Return the class name if the class is a parent and has only one child.
;;--------------------------------------------------------------------------
(clos:defgeneric two-subclass-check (c)
    (:documentation "Return the class name if it only has one subclass.")
    (:method ((c generic-class))
              (let ((result '()))
                   (dolist (one-class *list-of-classes*)
                             (if (member (name c) (inheritance one-class))
                                (push (name one-class) result)))
                   (if (= 1 (length result))
                      (name c)
                      '()) )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Whole/part evaluation
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;--------------------------------------------------------------------------
;; Takes one whole-part relation of a class and the class name and checks
;; to ensure the whole-part structure exists in the other class of the
;; structure and is the same.  If there is not corresponding structure,
;; "not-exist" is returned.  If there is more than one matching structure, or
;; the matching structure is not exactly the same, the structure(s) in the
;; other class is returned.
;;--------------------------------------------------------------------------
(defun check-wp (r cn)
 (typep r 'relation)
 (symbolp cn)
 (if (and (member (relation-class1 r) (class-namel))
          (member (relation-class2 r) (class-namel)))
    (let* ((other-wp '())
           (other-class (if (eql (relation-class1 r) cn)
                             (relation-class2 r)
                             (relation-class1 r)))
           (other-wp-list (whole-part (return-class other-class))))
         (dolist (one-wp other-wp-list)
                (if (or (eql (relation-class1 one-wp) cn)
                        (eql (relation-class2 one-wp) cn))
                   (setf other-wp (cons one-wp other-wp)) ))
         (cond ((null other-wp) 'not-exist)
              ((eql (length other-wp) 1)
               (if (rel-equalp r (first other-wp)) '()))
              (t other-wp)) )
    ;; else
    (if (not (member (relation-class1 r) (class-namel)))
          (relation-class1 r)
          (relation-class2 r)) ))

;;---------------------------------------------------------------------------
;; Returns the relation if the two classes in the relation are the same.
;;---------------------------------------------------------------------------
(defun relation-classes-different (a-relation)
 (typep a-relation 'relation)
 (if (eql (relation-class1 a-relation)
          (relation-class2 a-relation))
    (a-relation)
    '()))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Relation evaluation
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;---------------------------------------------------------------------------
;; Takes one relation of a class and the class name and checks to ensure the
;; relation exists in the oether class of the relations and the contents of the
;; slots are the same.  If there is no corresponding structure, "not-exist" is
;; returned.  If there is more than one matching structure, or the matching
;; structure is not excatly the same, the structure(s) in the other class
;; is returned.
;;---------------------------------------------------------------------------
(defun check-rel (r cn)
 (typep r 'relation)
 (symbolp cn)
 (if (and (member (relation-class1 r) (class-namel))
          (member (relation-class2 r) (class-namel)))
    (let* ((other-r '())
           (other-class (if (eql (relation-class1 r) cn)
                            (relation-class2 r)
                            (relation-class1 r)))
           (other-r-list (relation (return-class other-class))))
       (dolist (one-r other-r-list)
               (if (or (eql (relation-class1 one-r) cn)
                       (eql (relation-class2 one-r) cn))
                  (setf other-r (cons one-r other-r)) ))
       (cond ((null other-r) 'not-exist)
             ((eql (length other-r) 1)
              (if (rel-equalp r (first other-r)) '()))
             (t other-r)))
    ;; else
    (if (not (member (relation-class1 r) (class-namel)))
          (relation-class1 r)
          (relation-class2 r)) ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Utilities
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun remove-null (l)
 (check-type l list)
 (cond ((null l) '())
        ((null (first l))
         (remove-null (rest l)))
        ((listp (first l))
         (cons (remove-null (first l)) (remove-null (rest l))))
        (t (cons (first l) (remove-null (rest l)))) ))

(defun rel-equalp (r1 r2)
 "Checks two relation structures to see if the contents of their
slots is the same."
 (typep r1 'relation)
 (typep r2 'relation)
 (and (eql (relation-name r1) (relation-name r2))
    (eql (relation-class1 r1) (relation-class1 r2))
    (eql (relation-range1 r1) (relation-range1 r2))
    (eql (relation-class2 r1) (relation-class2 r2))
    (eql (relation-range2 r1) (relation-range2 r2))))
```

OAKSMOD.LISP

Model Modification Code

```
(in-package 'oaks)

(proclaim '(optimize (compilation-speed 3)))

;;*****************************************************************************
;; Any messages printed by oaksmod.lisp are to be printed using message boxes.
;; The print function is redefined in OAKS so that occurs.
;;*****************************************************************************
(shadow 'print)

(defun print (a-string)
  (stringp a-string)
  (create-error-message a-string))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; OAKS Model Modification Procedures
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; list of tests that need to run successfully before the model is complete.
(defvar *pending-issues* '())
(setf *pending-issues* (list '(classes need verified)))

;; list if tests that are advisory tests. These tests do not have to be resolved
;; before the model os complete, but should be brought to the user's attention.
(defvar *advisory-issues* '())

;;*****************************************************************************
;; Change the name of a class.
;;   The name of a class can change if the class name is unique within the model.
;; When a class name changes, the model must be searched for uses of the old class
;; name in attributes and services of other classes and the use of the name in
;; inheritance, whole-part and relation slots.
;;   The pending-issues list is affected. The class name must be changed in the entries
;; in the list. Also, there may be pending issues that are resolved due to a change in
;; class name that must be removed from the list. One of these is a pending issue that
;; is a relation tied to a class that is not in the model. If the class name changes
;; to the missing class, the relation is added to the class and the entry in pending
;; issues is removed.
;;*****************************************************************************
(defun change-class-name (old-name new-name)
  (symbolp old-name)
  (symbolp new-name)
  (cond ((member new-name (class-namel))
         (print "The name is already the name of an existing class."))
        ((not (member old-name (class-namel)))
         (print "The old class name does not exist in the model."))
        (t (progn
            (change-class-name-slot old-name new-name)
            (change-name-in-class old-name new-name)
            (change-class-in-inheritance old-name new-name)
            (change-class-in-relations old-name new-name)
            ;; change the class name in the pending-issues list
            (dolist (one-entry *pending-issues*)
                    (change-class-name-in-pending old-name new-name one-entry))
            ;; change the class name in the necessary-classes list
            (if (assoc old-name *necessary-classes*)
          (setf (first (assoc old-name *necessary-classes*)) new-name))
      (setf *pending-issues*
           (new-class-pending *pending-issues*))
      ;; add any relations waiting for the new class name
      (add-rel-to-new-class new-name))) ))

;;___________________________________________________________________________
;; Change the class name in the name slot of the class.
;;___________________________________________________________________________
(defun change-class-name-slot (old new)
 (symbolp old)
 (symbolp new)
 (setf (name (return-class old)) new))

;;___________________________________________________________________________
;; Change the name in the services and attributes of other classes, if used.
;; The name may be same as a local attribute, so each attribute and service must
;; be examined for external class names only.
;;___________________________________________________________________________
(defun change-name-in-class (old new)
 (symbolp old)
 (symbolp new)
 (dolist (one-class *list-of-classes*)
        (change-name-in-atts one-class old new)
        (change-name-in-servs one-class old new)))

;;___________________________________________________________________________
;; Change the class name if it is used in an attribute.
;;___________________________________________________________________________
(clos:defgeneric change-name-in-atts (c old new)
   (:documentation "Change the class name from old to new if used in an attribute.")
   (:method ((c generic-class) (old symbol) (new symbol))
          (dolist (one-att (state-space c))
                 (if (listp (attrs-base (a-set one-att)))
                      (dolist (one-attrs (attrs-base (a-set one-att)))
                            (class-name-att-sub one-attrs old new))
                      (class-name-att-sub (a-set one-att) old new)) )))

(defun class-name-att-sub (ma old new)
 (typep ma 'attrs)
 (if (or (eql (attrs-base ma) 'class)
        (eql (attrs-base ma) 'attrib))
    (if (eql (attrs-lower ma) old)
        (setf (attrs-lower ma) new) )))

;;___________________________________________________________________________
;; Change the class name if it is used in the services
;;___________________________________________________________________________
(clos:defgeneric change-name-in-servs (c old new)
   (:documentation "Change the external class name from old to new if used
in any service.")
   (:method ((c generic-class) (old symbol) (new symbol))
          (dolist (one-serv (services c))
                 (change-class-io-set (input-set one-serv) old new)
                 (change-class-io-set (output-set one-serv) old new)
                 (change-class-post (post one-serv) old new)) ))
```

```
(defun change-class-io-set (ios old new)
 (if (null ios) '()
    (progn
         (change-class-io-parameter (first ios) old new)
         (change-class-io-set (rest ios) old new))))

(defun change-class-io-parameter (p old new)
 (typep p 'parameterf)
 (change-class-para-values (parameterf-values p) old new))

(defun change-class-para-values (val old new)
 (cond ((null val) '())
         ((atom val) '())
         ((equal (first val) ':c)
          (if (eql (second val) old)
             (setf (second val) new)))
         ((equal (first val) ':a)
          (if (eql (second val) old)
             (setf (second val) new)))
         (t (progn
             (change-class-para-values (first val) old new)
             (change-class-para-values (rest val) old new))) ))

(defun change-class-post (post old new)
 (if (null post) '()
    (dolist (one-message (postf-messages post))
            (if (eql (first one-message) old)
                  (setf (first one-message) new)) )))

;;____________________________________________________________________________
;; Change the name of the class if it is used in the inheritance slot of other classes.
;;____________________________________________________________________________
(defun change-class-in-inheritance (old new)
 (symbolp old)
 (symbolp new)
 (dolist (one-class *list-of-classes*)
           (if (not (eql (name one-class) new))
              (setf (inheritance one-class)
                     (subst* new old (inheritance one-class))))))

;;____________________________________________________________________________
;; Change the name of the class if it is used in the whole-part or relations slots
;; of other classes.
;;____________________________________________________________________________
(defun change-class-in-relations (old new)
 (symbolp old)
 (symbolp new)
 (dolist (one-class *list-of-classes*)
           (change-class-in-rel (whole-part one-class) old new)
           (change-class-in-rel (relation one-class) old new)) )

(defun change-class-in-rel (rel-list old new)
 (listp rel-list)
 (symbolp old)
 (symbolp new)
 (dolist (one-rel rel-list)
        (if (eql (relation-class1 one-rel) old)
           (setf (relation-class1 one-rel) new))
        (if (eql (relation-class2 one-rel) old)
           (setf (relation-class2 one-rel) new)) ))
;;_______________________________________________________________________________
;; Change a class name in the pending issues list.
;;  All class names that match the old class name must be changed to the new name.
;; Care must be taken not to change an attribute or service that has the same name
;; as the old class name.
;;_______________________________________________________________________________
(defun change-class-name-in-pending (old new one-entry)
 (symbolp old)
 (symbolp new)
 (listp one-entry)
 (if (eql (second one-entry) old)
    (setf (second one-entry) new))
 (cond ((eql (first one-entry) 'check-parameter)
         (change-class-io-parameter (fifth one-entry) old new))
        ((eql (first one-entry) 'check-messages)
         (if (eql (first (fourth one-entry)) old)
            (setf (first (fourth one-entry)) new)))
        ((eql (first one-entry) 'missing-class-and-relation)
         (let ((rel (list (third one-entry))))
          (change-class-in-rel rel old new))) ))

;;*************************************************************************************
;; Change the description of a class.
;;*************************************************************************************
(defun change-class-desc (class-name new-desc)
 (symbolp class-name)
 (stringp new-desc)
 (setf (desc (return-class class-name)) new-desc))

;;*************************************************************************************
;; Change an attribute name
;;  The attribute name must be unique within the class and must not be one of the legal
;; set of attribute sets.
;;  The attribute name must change locally, in the name slot and in the services.
;;  The attribute name must change globally, in the attributes and services of other
;; classes.
;;  The attribute name is changed where it is used in pending issues.
;;  Any pending-issues entries resolved by the change of the attribute name are
;; removed.
;;*************************************************************************************

;;_______________________________________________________________________________
;; Change the name of an attribute given the class name, the old attribute name. and
;; the new attribute name.  The name is checked to ensure it is unique in the class
;; and is not a member of the legal sets of attributes.  Then the name is changed
;; in the local name slot, the attributes and services of other classes, and the
;; local services.
;;_______________________________________________________________________________
(defun change-att-name (class-name old-name new-name)
 (symbolp class-name)
 (symbolp old-name)
 (symbolp new-name)
 (cond ((member new-name (att-names (return-class class-name)))
        (print "The new name is the name of an existing attribte."))
       ((proper-attr-setp new-name)
        (print "The new name is the same as one of the legal sets for attributes."))
       ((not (member old-name (att-names (return-class class-name))))
        (print "The old name does not exist in the class."))
       (t (progn
           ;; change the local name slot
           (setf (name (return-attribute class-name old-name)) new-name)
           (dolist (one-class *list-of-classes*)
                   (change-att-name-in-atts one-class class-name old-name new-name)
                   (change-att-name-in-servs
                    one-class class-name old-name new-name)
                   ;; change attribute name ion the pending issues list
                   (dolist (one-entry *pending-issues*)
                           (change-att-name-in-pending
                            class-name old-name new-name one-entry))
                   ;; change the attribute name in the necessary classes list
                   (if (assoc class-name *necessary-classes*)
                       (setf (second (assoc class-name *necessary-classes*))
                             (substitute new-name old-name
                                         (second (assoc class-name *necessary-classes*)))))
                   ;; delete any resolved pending issues
                   (setf *pending-issues*
                         (remove-missing-att-entries *pending-issues*)) )))))

;;________________________________________________________________________
;; Change the name of an attribute of a class in the attributes of other classes.
;; The class of the changed attribute is needed because there may be an attribute
;; with the same name in a different class that is not to change.
;;________________________________________________________________________
(clos:defgeneric change-att-name-in-atts (c att-class old new)
   (:documentation "Change the name of an attribute from att-class from old
to new in the attributes of other classes.")
   (:method ((c generic-class)(att-class symbol)(old symbol)(new symbol))
           (dolist (one-att (state-space c))
                   (if (listp (attrs-base (a-set one-att)))
                       (dolist (one-attrs (attrs-base (a-set one-att)))
                               (att-name-sub one-attrs att-class old new))
                       (att-name-sub (a-set one-att)
                                     att-class old new) ))))

(defun att-name-sub (ma att-class old new)
 (typep ma 'attrs)
 (if (eql (attrs-base ma) 'attrib)
   (if (and (eql (attrs-lower ma) att-class)
            (eql (attrs-upper ma) old))
       (setf (attrs-upper ma) new) )))

;;________________________________________________________________________
;; Change an attribute name of a class in the services of its own class and those of
;; other classes. The attribute can be used in its own services in the input-set,
;; output-set, pre and post. The attribute can be used in other classes in the
;; input-set and output-set. The pre will only contain information on local state.
;; The atts slot of the post contain info on local attributes and new values. The
;; messages slot of post does not use any attributes.
;; c is the class where the attribute is being changed.
;; att-class is the name of the class the attribute belongs to.
;; old is the old attribute name.
;; new is the new attribute name.
;;_______________________________________________________________________
(clos:defgeneric change-att-name-in-servs (c att-class old new)
   (:documentation "Change an attribute name in the services of a class.")
   (:method ((c generic-class)(att-class symbol)(old symbol)(new symbol))
          (dolist (one-serv (services c))
                (change-att-io-set (input-set one-serv) c att-class old new)
                (change-att-io-set (output-set one-serv) c att-class old new)
                (if (eql (name c) att-class)
                     (progn
                      (setf (pre one-serv) (subst* new old (pre one-serv)))
                      (change-att-post (post one-serv) old new))) )))

;;_______________________________________________________________________
;; Change an attribute in the input-set or output-set of a service.  The attribute
;; name would be in the values slot, either as a local attribute or an external
;; attribute.  If it external, the attribute must be of class att-class, because
;; there can be the same name for attributes of different classes.
;; ios is the input- or output-set.
;; att-class is the class of the attribute to be changed.
;; old is the old attribute name.
;; new is the new attribute name.
;;_______________________________________________________________________
(defun change-att-io-set (ios c att-class old new)
 (if (null ios) '()
   (dolist (one-p ios)
          (cond ((null (parameterf-values one-p)) '())
                ((atom (parameterf-values one-p))
                 (if (eql (name c) att-class)
                      (if (eql (parameterf-values one-p) old)
                         (setf (parameterf-values one-p) new))))
                ((eql (first (parameterf-values one-p)) ':a)
                 (if (and (eql (second (parameterf-values one-p)) att-class)
                          (eql (third (parameterf-values one-p)) old))
                      (setf (third (parameterf-values one-p)) new)))
                (t '()) ))))

;;_______________________________________________________________________
;; Change the name of a local attribute in the post of a local service.  The atts
;; slot of post is examined for the name of the local attribute and it is changed if
;; found.
;;_______________________________________________________________________
(defun change-att-post (post old new)
 (if (null post) '()
   (if (null (postf-atts post)) '()
        (dolist (one-att-val (postf-atts post))
               (if (eql (attr-val-name one-att-val) old)
                  (setf (attr-val-name one-att-val) new))
               (if (listp (attr-val-value one-att-val))
                  (setf (attr-val-value one-att-val)
                        (subst* new old (attr-val-value one-att-val)))) ))))
```

```
;;___________________________________________________________________________
;; Change the name of the attribute where used in pending-issues
;;___________________________________________________________________________
(defun change-att-name-in-pending (cn old new one-entry)
 (symbolp cn)
 (symbolp old)
 (symbolp new)
 (listp one-entry)
 (cond ((or (eql (first one-entry) 'atts-classc)
            (eql (first one-entry) 'atts-attc)
            (eql (first one-entry) 'null-a-set))
         (if (and (eql (second one-entry) cn)
                  (eql (third one-entry) old))
            (setf (third one-entry) new)))
        ((eql (first one-entry) 'check-parameter)
         (change-att-io-set (list (fifth one-entry)) (return-class (second one-entry))
                            cn old new))
        ((eql (first one-entry) 'check-attr-val)
         (if (and (eql (second one-entry) cn)
                  (eql (fourth one-entry) old))
            (setf (fourth one-entry) new))) ))

;;*******************************************************************************
;; Change the description of an attribute.
;;*******************************************************************************
(defun change-att-desc (class-name att-name new-desc)
 (symbolp class-name)
 (symbolp att-name)
 (stringp new-desc)
 (setf (desc (return-attribute class-name att-name)) new-desc))

;;*******************************************************************************
;; Change the initial value of an attribute
;;*******************************************************************************
(defun change-initial-value (class-name att-name new-initial-value)
 (symbolp class-name)
 (symbolp att-name)
 (if (listp new-initial-value)
    (setf (initial-value (return-attribute class-name att-name)) new-initial-value)
   (print "The initial value must be a list")))

;;*******************************************************************************
;; Change the a-set slot of an attribute.
;; The input is a base and an optional lower and upper value.  If the base is a list,
;; there is no lower or upper value.  If the base is a list, each element of the
;; list consists of sublists made of a base, and an optional lower and upper value.
;;*******************************************************************************

;;___________________________________________________________________________
;; This procedure requires that checking of the structure be done before
;; entering, or else the procedure will not run.  If the base value is a list,
;; the base values of the sublist must satisfy proper-attr-setp.  If the base value
;; is an atom, it must satisfy proper-attr-setp.
;; If the base value is "class" there must be a lower value.  If the base value
;; is "enum", there must be a lower value.  If the base value is "attrib", there
;; must be a lower and upper value.
```

```
;;________________________________________________________________________
(defun change-attr-a-set (class-name att-name base-value &optional
                                          (lower-value '()) (upper-value '()))
 (symbolp class-name)
 (symbolp att-name)
 (let ((the-att (return-attribute class-name att-name))
        (result '()))
  ;; if 1
  (if (listp base-value)
        ;; then 1
        (progn
          (dolist (one-element base-value)
                  (let ((one (first one-element))
                        (two (if (> (length one-element) 1)
                                 (second one-element)
                              '()))
                        (three (if (> (length one-element) 2)
                                   (third one-element)
                                   '())))
                    ;; if one is not correct, do not change the attribute structure.
                    ;; if 2
                    (if (null (create-attrs-structure one two three))
                        ;; then 2
                        (progn
                          (setf result '())
                          (return))
                      ;; else 2
                      (setf result (cons (create-attrs-structure one two three)
                                         result))) ))
          (if (not (null result))
            (setf (a-set the-att) (make-attrs :base result))) )
    ;; else 1
    (progn
          (setf result (create-attrs-structure base-value lower-value upper-value))
          (if (not (null result))
            (setf (a-set the-att) result)) ))
  ;; Do the system wide checks based on the changes.
  ;; if 3
  (if (not (null result))
        ;; then 3
        (let* ((class-check (atts-classc the-att))
               (att-check (atts-attc the-att)))
          (if class-check
            (progn
                (print "Some classes aren't valid")
                (setf *pending-issues*
                      (cons (list 'atts-classc class-name att-name) *pending-issues*)))
            ;; if the test passes, remove it from the pending issues if it is there.
            (setf *pending-issues*
                    (remove
                     (list 'atts-classc class-name att-name) *pending-issues*
                     :test #'equal)))
          ;; if the class check fails, the class for the attribute may not be valid.
          ;; when the class check passes, the attribute check should then be run, so
          ;; it is automatically put on the list of pending tests
          (if att-check
          (progn
              (print "Some attributes aren't valid")
              (setf *pending-issues*
                  (cons (list 'atts-attc class-name att-name) *pending-issues*)))
          (setf *pending-issues*
                (remove (list 'atts-attc class-name att-name) *pending-issues*
                        :test #'equal)))
       (setf *pending-issues*
             (remove (list 'null-a-set class-name att-name) *pending-issues*
                     :test #'equal))
       (setf *pending-issues*
             (remove-duplicates *pending-issues* :test #'equalp)) ))))

(defun create-attrs-structure (base-value &optional (lower-value '()) (upper-value '()))
 (cond ((eql base-value 'enum)
        (if (not (null lower-value))
          (make-attrs :base base-value
                      :lower lower-value)
          (progn
            (print "Enum structure not specified.")
            '() )))
       ((eql base-value 'class)
        (if (not (null lower-value))
          (make-attrs :base base-value
                      :lower lower-value)
          (progn
            (print "Class not specified.")
            '() )))
       ((eql base-value 'attrib)
        (if (and (not (null lower-value))
                 (not (null upper-value)))
          (make-attrs :base base-value
                      :lower lower-value
                      :upper upper-value)
          (progn
            (print "Attribute not specified.")
            '() )))
       (t
        (if (proper-attr-setp base-value)
          (make-attrs :base base-value
                      :lower lower-value
                      :upper upper-value)
         (progn
          (print "Base does not satisfy attribute structure values.")
          '()))) ))

;;*****************************************************************************************
;; Delete an attribute from a class.
;;   Pending issues for that attribute are removed.  Pending issues are added for the
;; services and attributes that reference the deleted attribute.
;;   The attribute of a class that is a parent cannot be deleted.
;;*****************************************************************************************
(defun delete-attribute (class-name attribute-name)
 (symbolp class-name)
 (symbolp attribute-name)
 (let ((the-class (return-class class-name)))
    (if (member attribute-name (att-names the-class))
         (if (not (member attribute-name (second (assoc class-name *necessary-classes*))))
           (let ((the-att (return-attribute class-name attribute-name)))
             (if (not (parentp the-class))
                  (progn
                    (setf (state-space the-class)
                            (remove the-att (state-space the-class) :test #'equalp))
                    ;; remove pending issues for that attribute
                    (setf *pending-issues*
                            (remove (list 'atts-classc class-name attribute-name)
                                   *pending-issues* :test #'equal))
                    (setf *pending-issues*
                            (remove (list 'atts-attc class-name attribute-name)
                                   *pending-issues* :test #'equal))
                    (setf *pending-issues*
                            (remove (list 'null-a-set class-name attribute-name)
                                   *pending-issues* :test #'equal))
                    ;; add to pending issues
                    (mapcar #'attr-del-check *list-of-classes*)
                    (setf *pending-issues*
                            (remove-duplicates *pending-issues* :test #'equalp)))
                  (print "The class is a parent so the attribute cannot be deleted.")))
           (print "The attribute cannot be deleted."))
      (print "The attribute is not part of the class.")) ))

;;____________________________________________________________________________
;; Check every class in the model for possible invalid attributes.  For each
;; invalid attribute found, add an entry to pending-issues.
;;____________________________________________________________________________
(defun attr-del-check (a-class)
 (dolist (one-attr (state-space a-class))
        (if (atts-classc one-attr)
           (setf *pending-issues*
                 (cons (list 'atts-classc (name a-class) (name one-attr))
                       *pending-issues*)))
        (if (atts-attc one-attr)
           (setf *pending-issues*
                 (cons (list 'atts-attc (name a-class) (name one-attr))
                       *pending-issues*))))
 (dolist (one-serv (services a-class))
        (dolist (input-para (input-set one-serv))
              (if (remove-null (check-parameter (name a-class) input-para))
                 (setf *pending-issues*
                       (cons (list 'check-parameter (name a-class) (name one-serv)
                                   'input-set input-para) *pending-issues*))))
        (dolist (output-para (output-set one-serv))
              (if (remove-null (check-parameter (name a-class) output-para))
                 (setf *pending-issues*
                       (cons (list 'check-parameter (name a-class) (name one-serv)
                                   'output-set output-para) *pending-issues*))))
        (if (post one-serv)
           (if (postf-atts (post one-serv))
              (dolist (one-attr-val (postf-atts (post one-serv)))
                     (if (check-attr-val one-attr-val (name a-class))
                        (setf *pending-issues*
                              (cons (list 'check-attr-val (name a-class)
                                                  (name one-serv)
                                                  (attr-val-name one-attr-val))
                                            *pending-issues*)))))) )
 (setf *pending-issues*
      (remove-duplicates *pending-issues* :test #'equalp)))

;;*******************************************************************************
;;
;; Add an attribute to a class.
;;   Remove all entries in pending-issues related to that attribute being missing.  If
;; the a-set portion of the new attribute is invalid, add an entry to pending-issues
;; and set the a-set slot to null.
;;*******************************************************************************
;;
(defun add-attribute (class-name att-name desc base-value &optional
                                    (lower-value '()) (upper-value '()))
 (symbolp class-name)
 (symbolp att-name)
 (stringp desc)
 (let ((the-class (return-class class-name)))
  (if (not (member att-name (att-names the-class)))
      (progn
       (setf (state-space the-class)
             (cons (make-instance 'attribute
                                  :name att-name
                                  :desc desc
                                  :a-set (make-attrs :base '())
                                  :verif t)
                   (state-space the-class)))
       (change-attr-a-set class-name att-name base-value lower-value upper-value)
       (if (not (attrs-base (a-set (return-attribute class-name att-name))))
         (progn
             (setf *pending-issues*
                   (cons (list 'null-a-set class-name att-name) *pending-issues*))
             (print "The a-set value was invalid and set to null.")))
       ;; get rid of pending-issues due to missing attribute
       (setf *pending-issues*
             (remove-missing-att-entries *pending-issues*)) )
    (print "The attribute name is already the name of an existing attribute."))))

;;
;;_______________________________________________________________________________
;; Run all the pending-issues entries involved with a missing attribute.  Remove any
;; entries where the test passes.
;;
;;_______________________________________________________________________________
(defun remove-missing-att-entries (pending)
 (cond ((null pending) '())
       ((eql (first (first pending)) 'atts-attc)
        (if (atts-attc (return-attribute (second (first pending))
                                         (third (first pending))))
          (cons (first pending) (remove-missing-att-entries (rest pending)))
          (remove-missing-att-entries (rest pending))))
       ((eql (first (first pending)) 'check-parameter)
        (if (remove-null (check-parameter (second (first pending))
                                          (fifth (first pending))))
          (cons (first pending) (remove-missing-att-entries (rest pending)))
          (remove-missing-att-entries (rest pending))))
       ((eql (first (first pending)) 'check-attr-val)
        (if (check-attr-val (make-attr-val :name (fourth (first pending))
                                    :value '()) (second (first pending)))
            (cons (first pending) (remove-missing-att-entries (rest pending)))
            (remove-missing-att-entries (rest pending))))
        (t (cons (first pending) (remove-missing-att-entries (rest pending)))) ))

;;*****************************************************************************
;;
;; Change the name of a service. The name must be changed in the name slot of the
;; service and in the message slot of the post of other classes. The name is also change
;; where used in pending-issues. Also, any pending-issues resolved by the name
;; change are removed.
;;*****************************************************************************
;;
(defun change-service-name (class-name old-name new-name)
 (symbolp class-name)
 (symbolp old-name)
 (symbolp new-name)
 (if (member new-name (ser-names (return-class class-name)))
    (print "The new name is the name of an existing service.")
    (progn
         (setf (name (return-service class-name old-name)) new-name)
         (dolist (one-class *list-of-classes*)
                 (if (not (eql (name one-class) class-name))
                    (change-ser-name-in-messages
                     one-class class-name old-name new-name)))
         ;; change name in pending issues
         (dolist (one-entry *pending-issues*)
                 (change-ser-name-in-pending class-name old-name new-name one-entry))
         ;; change name in necessary classes
         (if (assoc class-name *necessary-classes*)
            (setf (third (assoc class-name *necessary-classes*))
                  (substitute new-name old-name
                              (third (assoc class-name *necessary-classes*)))))
         ;; delete entries in pending issues
         (setf *pending-issues*
              (remove-missing-serv-entries *pending-issues*)) )))

;;_____________________________________________________________________________
;; Change the service name in the message slot of the post of services of other
;; classes.
;;_____________________________________________________________________________
(clos:defgeneric change-ser-name-in-messages (c serv-class old new)
    (:documentation "Change a service name in the message slot of the
postconditions of services of other classes.")
    (:method ((c generic-class)(serv-class symbol)(old symbol)(new symbol))
             (dolist (one-serv (services c))
                     (if (not (null (post one-serv)))
                         (if (not (null (postf-messages (post one-serv))))
                            (dolist (one-message (postf-messages (post one-serv)))
                                    (if (and (eql (first one-message) serv-class)
                                             (eql (second one-message) old))
                                        (setf (second one-message) new)) ))))))

;;_____________________________________________________________________________
;; Change the name of a service in an entry of pending-issues.
;;_____________________________________________________________________________
(defun change-ser-name-in-pending (cn old new one-entry)
 (symbolp cn)
 (symbolp old)
 (symbolp new)
 (listp one-entry)
 (cond ((or (eql (first one-entry) ':service)
               (eql (first one-entry) 'check-parameter)
               (eql (first one-entry) 'check-attr-val))
          (if (and (eql (second one-entry) cn)
                     (eql (third one-entry) old))
               (setf (third one-entry) new)))
         ((eql (first one-entry) 'check-messages)
          (if (and (eql (second one-entry) cn)
                     (eql (third one-entry) old))
               (setf (third one-entry) new)
               (if (and (eql (first (fourth one-entry)) cn)
                          (eql (second (fourth one-entry)) old))
                    (setf (second (fouth one-entry)) new))))
         (t '())))

;;*******************************************************************************
;; Change the description of a service.
;;*******************************************************************************
(defun change-serv-desc (class-name serv-name new-desc)
 (symbolp class-name)
 (symbolp serv-name)
 (stringp new-desc)
 (setf (desc (return-service class-name serv-name)) new-desc))

;;*******************************************************************************
;; Change a parameter of the input set of a service.
;;  The input parameters are:
;;  1. class-name
;;  2. service-name
;;  3. old-name-val-list. To add an input parameter, this list is set to (*add).
;; Otherwise, it is a list consisting of the name and the values of the parameter
;; to either delete or replace (name values)
;;  4. new-name-val-list. To delete an input parameter, this list is set to
;; (*delete). Otherwise, it is a list consisting of the name and values of a new
;; parameter that either replaces an existing one or is added to the input-set.
;;*******************************************************************************
(defun change-input-set (class-name service-name old-name-val-list new-name-val-list)
 (symbolp class-name)
 (symbolp service-name)
 (listp old-name-val-list)
 (listp new-name-val-list)
 (let* ((the-class (return-class class-name))
          (the-service (return-service class-name service-name))
          (the-input-set (input-set the-service)))
   (cond ((eql (first old-name-val-list) '*add)
            ;; add a new parameter to the input set
            (if (eql (length new-name-val-list) 2)
               ;; if 1
               ;; the new list must contain two elements
               (if (unique-para-name the-input-set the-class (first new-name-val-list))
                     ;; if 2
                     ;; the name of the new parameter must be unique
                     (add-to-io-set new-name-val-list the-class the-service)
```

```
                ;; else 2
                (print "The name of the new parameter was not unique."))
          ;; else 1
          (print "The new parameter list did not contain two elements")) )
      ((eql (first new-name-val-list) '*delete)
       ;; delete a parameter from the input set
       (if (in-para-list old-name-val-list the-input-set)
          ;; the parameter must be in the list
          (remove-val class-name service-name old-name-val-list)
          (print "The old list does not exist in the parameter list")))
      (t
       ;; the old list is replaced by the new list
       (if (eql (length old-name-val-list) 2)
          ;; if 3
          ;; the old list must contain two elements
          (if (eql (length new-name-val-list) 2)
                ;; if 4
                ;; the new list must contain two elements
                (if (in-para-list old-name-val-list the-input-set)
                   ;; if 5
                   ;; the parameter must be in the list
                   (progn
                        (if (not (eql (first old-name-val-list)
                                      (first new-name-val-list)))
                           ;; if 6, no else
                           ;; replace the parameter name
                           (replace-name-in-io-set the-class the-service
                                                   the-input-set old-name-val-list
                                                   new-name-val-list))
                        (if (not (equal (second old-name-val-list)
                                        (second new-name-val-list)))
                           ;; if 7, no else
                           ;; replace the parameter value
                           (replace-value-in-io-set the-class the-service the-input-set
                                                    old-name-val-list new-name-val-list)))
                   ;; else 5
                   (print "The old values do not exist."))
                ;; else 4
                (print "The new value list does not contain two elements."))
          ;; else 3
          (print "The old value list does not contain two elements.")) ))))

;;____________________________________________________________________________
;; Add a new parameter to an io set of a service. The values of the new parameter
;; are checked to ensure the class and attributes, if any, used are valid. If not,
;; they are added to the list of pending issues. Also, the pending issues list is
;; examined to see if the name of the new parameter will eliminate a pending issue
;; that was generated when a parameter was deleted, and that parameter name was in
;; the pre or post conditions. This owuld happen if the name was the same as the
;; parameter name that was deleted.
;;____________________________________________________________________________
(defun add-to-io-set (new c s)
  (if (first new)
     (setf (input-set s)
           (cons (make-parameterf :name (first new)
                                  :values (second new))
                (input-set s)))
     (setf (output-set s)
           (cons (make-parameterf :name (first new)
                                  :values (second new))
                (output-set s))))
   (check-classes-and-atts (name c) (name s) '(*add) new)
   (check-io-name-pending (name c) (name s) (first new)))

;;______________________________________________________________________________
;; Delete a parameter from the io sets of a service. If a parameter is deleted, and
;; its name is  used in the pre or postconditions, information is added to the pending
;; issues lits that says that parameter name is no longer valid.  The pending issues
;; entry takes the form of:
;;   (:service class-name service-name pre missing input-set deleted-parameter-name)
;;   (:service class-name service-name post missing input-set deleted-parameter-name)
;; The name being deleted is not applicable to the output set - only the input set.
;; If a parameter is deleted that has a pending issue doe to an invalid attribute
;; or class in its values, remove that issue from the pending issue list.
;;______________________________________________________________________________
(defun remove-val (cn sn old)
  (symbolp cn)
  (symbolp sn)
  (listp old)
  (let ((the-service (return-service cn sn)))
    (if (first old)
        (setf (input-set the-service)
              (remove (make-parameterf :name (first old)
                                       :values (second old))
                      (input-set the-service) :test #'equalp))
      (setf (output-set the-service)
              (remove (make-parameterf :name (first old)
                                       :values (second old))
                      (output-set the-service) :test #'equalp)))
    ;; if a parameter is deleted that has a pending issue due to the
    ;; value not passing the check-parameter test (test of the existance of any
    ;; attributes and values used), remove the issue from pending issues.
    (check-classes-and-atts cn sn old '(*delete))
    ;; If the input-set name is used in the pre or post, add an entry to pending
    ;; issues because the name is no longer valid.
    (if (first old)
        (progn
          (if (pre the-service)
              (if (any-member (first old) (pre the-service))
                  (setf *pending-issues*
                        (cons (list ':service cn sn 'pre 'missing 'input-set (first old))
                              *pending-issues*))))
          (if (post the-service)
              (if (postf-atts (post the-service))
                  (if (any-member (first old) (postf-atts (post the-service)))
                      (setf *pending-issues*
                            (cons (list
                                   ':service cn sn 'post 'missing 'input-set (first old))
                                  *pending-issues*)))))
          (setf *pending-issues*
                (remove-duplicates *pending-issues* :test #'equalp)))) ))
```

```
;;______________________________________________________________________
;; Replace the name in a parameter of an input-set of a service.
;; First check to ensure the name is unique.  If it is, change the name in the name
;; slot of the parameter and anywhere it is used in the pre and postconditions.
;; Also, check the pending issues list to see if there is an entry for a missing
;; parameter name in the pre or post.
;;______________________________________________________________________
(defun replace-name-in-io-set (c s is old new)
  (if (unique-para-name is c (first new))
     (progn
        (dolist (one-para is)
              (if (and (eql (parameterf-name one-para) (first old))
                      (equal (parameterf-values one-para) (second old)))
                 (setf (parameterf-name one-para) (first new))))
        (check-pre-and-post s (first old) (first new))
        (check-io-name-pending (name c) (name s) (first new)))
     (print "The new parameter name is already the name of an input parameter
or an attribute or a legal attribute set.")))

;;______________________________________________________________________
;; Replace the value of a parameter of an input set of a service.
;; First find the input set that matches the old input set and replace the old value
;; with the new.  Then ensure the values are legal - if not, add to pending issues.
;; Also remove any pending isues to do with the value of the old input set.
;;______________________________________________________________________
(defun replace-value-in-io-set (c s is old new)
  (dolist (one-para is)
          (if (and (eql (parameterf-name one-para) (first old))
                  (equal (parameterf-values one-para) (second old)))
             (setf (parameterf-values one-para) (second new))))
  (check-classes-and-atts (name c) (name s) old new))

;;______________________________________________________________________
;; Check a new parameter name to ensure it is not the name of an existing parameter,
;; attribute, or the set of legal attribute values.
;;______________________________________________________________________
(defun unique-para-name (io-set class new-name)
  (not (or (member new-name (mapcar #'parameterf-name io-set))
           (member new-name (att-names class))
           (proper-attr-setp new-name))))

;;______________________________________________________________________
;; Check to ensure any external classes and local or external attributes used in the
;; values slot of the input/output parameter are valid.  If not, place an entry in
;; pending issues.  If the value slot is being replaced, remove any entries in
;; pending issues for the old information in the values slot.
;;______________________________________________________________________
(defun check-classes-and-atts (cn sn old-para new-para)
  (if (eql (length new-para) 2)
     (if (check-parameter cn (make-parameterf :name (first new-para)
                                              :values (second new-para)))
        (progn
          (print "The values have invalid references.")
          (if (first new-para)
              (setf *pending-issues*
                 (cons (list 'check-parameter cn sn 'input-set new-para)
                                        *pending-issues*))
            (setf *pending-issues*
                  (cons (list 'check-parameter cn sn 'output-set new-para)
                        *pending-issues*))))))
  ;; Check to see if the old parameter has any entries in pending issues.  If so,
  ;; remove them.
  (if (not (eql (first old-para) '*add))
     (if (first old-para)
          (setf *pending-issues*
                (remove (list 'check-parameter cn sn 'input-set old-para)
                        *pending-issues* :test #'equalp))
          (setf *pending-issues*
                (remove (list 'check-parameter cn sn 'output-set old-para)
                        *pending-issues* :test #'equalp))))
  (setf *pending-issues*
         (remove-duplicates *pending-issues* :test #'equalp)) )

;;_______________________________________________________________________________
;; Check to see if a (name value) pair are members of the input-set or output-set.
;;_______________________________________________________________________________
(defun in-para-list (val-list io-set)
  (let ((result '()))
    (dolist (one-para io-set result)
          (if (and (eql (parameterf-name one-para) (first val-list))
                   (equal (parameterf-values one-para) (second val-list)))
              (return t) ))))

;;_______________________________________________________________________________
;; Check the pre and post condition to see if the old parameter name is used.  If so,
;; replace with the new parameter name.
;;_______________________________________________________________________________
(defun check-pre-and-post (s old new)
  (if (pre s)
     (setf (pre s)
           (subst* new old (pre s))))
  (if (post s)
     (if (postf-atts (post s))
         (dolist (one-attr-val (postf-atts (post s)))
               (if (listp (attr-val-value one-attr-val))
                  (setf (attr-val-value one-attr-val)
                        (subst* new old (attr-val-value one-attr-val))) )))))

;;_______________________________________________________________________________
;; Check to see if the new name of a parameter matches a missing parameter name in
;; one of the pending issues.  If so, delete the entrie(s) in pending issues.
;;_______________________________________________________________________________
(defun check-io-name-pending (cn sn new-name)
  (setf *pending-issues*
        (remove (list ':service cn sn 'pre 'missing 'input-set new-name)
                *pending-issues* :test #'equalp))
  (setf *pending-issues*
        (remove (list ':service cn sn 'post 'missing 'input-set new-name)
                *pending-issues* :test #'equalp)))

;;*****************************************************************************
;;
;; Change the output set of a service by adding, deleting, or changing one parameter
;; in the output set.  The parameters of the output set have null as their name
;; and either a local attribute, another class, the attribute of another class,
;; a legal attribute set, or a list of any of the above.  If a parameter is added
;; or changed, the new values are evaluated to see if they are legal.  If not, an
;; entry is added to pending issues.
;;*************************************************************************************
(defun change-output-set (class-name service-name old-values new-values)
 (symbolp class-name)
 (symbolp service-name)
 (let* ((the-class (return-class class-name))
          (the-service (return-service class-name service-name))
          (the-output-set (output-set the-service)))
   (cond ((eql old-values '*add)
           ;; add a new parameter to the output set
           (add-to-io-set (list '() new-values) the-class the-service))
          ((eql new-values '*delete)
           ;; delete a parameter from the output set
           (if (in-para-list (list '() old-values) the-output-set)
              (remove-val class-name service-name (list '() old-values))
              (print "The old values do not exist in the parameter list")))
          (t
           ;; replace a parameter
           (if (in-para-list (list '() old-values) the-output-set)
              (replace-value-in-io-set the-class the-service the-output-set
                                          (list '() old-values) (list '() new-values))
              (print "The old values do not exist in the parameter list."))) )))

;;*************************************************************************************
;; Change the precondition of a service.  The precondition is a free-form list that
;; evaluates to true or false.  The new precondition replaces the existing precondition.
;; If there is an existing pre, check pending-issues to see if there is an entry for
;; an invalid parameter name in the old pre.  If the name is not used in the new
;; pre, delete this entry from pending issues.
;;*************************************************************************************
(defun change-serv-pre (class-name service-name new-pre)
 (symbolp class-name)
 (symbolp service-name)
 (listp new-pre)
 (dolist (one-entry *pending-issues*)
          (if (eql (first one-entry) ':service)
             (if (and (eql (second one-entry) class-name)
                        (eql (third one-entry) service-name)
                        (eql (fourth one-entry) 'pre)
                        (eql (fifth one-entry) 'missing))
                   (progn
                    (if (not (any-member (seventh one-entry) new-pre))
                        (setf *pending-issues*
                              (remove one-entry *pending-issues* :test #'equalp)))
                    (return) ))))
 (setf (pre (return-service class-name service-name)) new-pre))

;;*************************************************************************************
;; Change the atts portion of the post of a service.
;;    old-list : a list consisting of the old name and the old value, or (*add) if
;; a new structure is being added.
;;    new-list : a list consisting of the new name and the new value, or (*delete)
;; if an existing structure is being deleted.
;;    The atts portion consists of a list (possibly empty) of attr-val structures.
;;    A new structure could be added, an existing structure deleted, or the name slot
;; and/or value slot of a structure replaced.  If the name slot is changed or a new
;; structure added, the name must be the name of an existing local attribute and
;; must not be the name of another structure within the atts portion of the post.
;; If the name slot is not the name of an existing attribute, an entry is added
;; to pending-issues indicating the attribute does not exist.
;; If a structure is deleted, or the attribute name of an existing structure is
;; changed, an pending-issues associated with an invalid attrubte name in the deleted
;; or old structure are removed.  If a structure is deleted or the value slot is
;; changed, an entries in pending-issues associated with the use of a missing
;; input parameter name are removed if:
;;    1. the name is not used in the new value
;;    2. the name is not used in the value slots of any of the other structures in
;; atts.
;;    When a new value is added, it is a free-form structure.  The input parameter list
;; is not checked because it is not required to use names from that list.
;;*************************************************************************************
(defun change-serv-post-atts (class-name service-name old-list new-list)
 (symbolp class-name)
 (symbolp service-name)
 (listp old-list)
 (listp new-list)
 (let* ((the-class (return-class class-name))
        (the-service (return-service class-name service-name))
        (the-post (post the-service)))
  (cond ((eql (first old-list) '*add)
         (if (eql (length new-list) 2)
            (if (unique-attr-atts the-post new-list)
                 (add-to-postf-atts the-class the-service new-list)
                 (print "The attribute already has an entry in the post."))
            (print "The new attribute-value list did not contain two elements.")))
        ((eql (first new-list) '*delete)
         (if (in-post-atts old-list the-post)
            (remove-from-post-atts the-class the-service old-list)
            (print "The old attribute-value list does not exist in post.")))
        (t
         (if (eql (length old-list) 2)
            (if (eql (length new-list) 2)
                 (if (in-post-atts old-list the-post)
                    (progn
                        (if (not (eql (first old-list) (first new-list)))
                           (if (unique-attr-atts the-post new-list)
                                 (replace-post-atts-attr the-class the-service old-list
                                                                    new-list)
                              (print "The new attribute has an entry in post.")))
                        (if (not (equal (second old-list) (second new-list)))
                           (replace-post-atts-value the-class the-service
                                                            old-list new-list)))
                    (print "The old attribute-value list does not exist in post."))
                 (print "The new attribute-value list does not contain two elements."))
            (print "The old attribute-value list does not contain two elements."))) )))

;;______________________________________________________________________________
;; Return true if the name of the attribute (the first of new) is not the name
;; of any attributes used in the atts of post.
;;_____________________________________________________________________
(defun unique-attr-atts (post new)
  (if post
    (if (postf-atts post)
        (if (any-member (first new) (mapcar #'attr-val-name (postf-atts post)))
          '()
          t)
        t)
    t))

;;_____________________________________________________________________
;; Add a new attr-val structure to the atts of the post of a service.
;;    c : the class
;;    s : the service
;;    new : a list consisting of the new attribute name and the value
;; check-post-atts-attr checks to ensure the new attribute is valid, and adds an
;; entry to pending-issues if it is not.
;;_____________________________________________________________________
(defun add-to-postf-atts (c s new)
  (if (post s)
    (setf (postf-atts (post s))
          (cons (make-attr-val :name (first new)
                               :value (second new))
                (postf-atts (post s))))
    (setf (post s)
          (make-postf :atts `(,(make-attr-val :name (first new)
                                              :value (second new))))))
  (check-post-atts-attr c s '*add new))

;;_____________________________________________________________________
;; Return true if attr-val structure exists in the atts of post of a service.
;;   old : a list consisting of an existing name and value
;;   post : the post of a service
;;_____________________________________________________________________
(defun in-post-atts (old post)
  (if post
    (if (postf-atts post)
        (let ((result '()))
          (dolist (one-attr-val (postf-atts post) result)
                  (if (and (eql (attr-val-name one-attr-val) (first old))
                           (equal (attr-val-value one-attr-val) (second old)))
                      (return t))))
        '())
    '()))

;;_____________________________________________________________________
;; Remove an existing attribute-value structure from the atts of the post of a
;; service.  Then call check-post-atts-attr to remove any entries in pending-issues
;; on an invalid attribute for that entry, if any.  Also call
;; check-post-atts-missing-input-para to remove any entries in pending-issues on
;; input parameter names used that are not valid.
;;_____________________________________________________________________
(defun remove-from-post-atts (c s old)
  (listp old)
  (setf (postf-atts (post s))
        (remove (make-attr-val :name (first old)
                               :value (second old))
                (postf-atts (post s)) :test #'equalp))
 (check-post-atts-attr c s old '*delete)
 (check-post-atts-missing-input-para c s old '*delete))

;;______________________________________________________________________________
;; Replace the name slot of an attr-val structure of atts of the post of a service.
;; Call check-post-atts-attr to determine if the attribute is valid and, if not, add
;; an entry to pending-issues.  Also, if the replaced attribute name had an entry
;; in pending-issues, remove it.
;;______________________________________________________________________________
(defun replace-post-atts-attr (c s old new)
 (dolist (one-attr-val (postf-atts (post s)))
        (if (and (eql (attr-val-name one-attr-val) (first old))
                 (equal (attr-val-value one-attr-val) (second old)))
            (setf (attr-val-name one-attr-val) (first new))))
 (check-post-atts-attr c s old new))

;;______________________________________________________________________________
;; Replace the value slot of an attr-val atructure of atts of the post of a service.
;; Call check-post-atts-missing-input-para to remove entries from pending-issues
;; involving input parameter names used by the old value that are no longer valid.
;;______________________________________________________________________________
(defun replace-post-atts-value (c s old new)
 (dolist (one-attr-val (postf-atts (post s)))
        (if (and (eql (attr-val-name one-attr-val) (first new))
                 ;; use the attr name of new because it may be changed already
                 (equal (attr-val-value one-attr-val) (second old)))
            (setf (attr-val-value one-attr-val) (second new))))
 (check-post-atts-missing-input-para c s old new))

;;______________________________________________________________________________
;; Check the validity of new attribute names in the name slot of the attr-val
;; structure of the atts slot of the post of a service.  Also, if an attribute name
;; is either being replaced or deleted, remove any entries in pending-issues for
;; that attribute,
;;______________________________________________________________________________
(defun check-post-atts-attr (c s old new)
 ;; check validity of new attribute names
 (if (not (eql new '*delete))
    (if (check-attr-val (make-attr-val :name (first new)
                                       :value (second new)) (name c))
        (progn
          (setf *pending-issues*
                (cons (list 'check-attr-val (name c) (name s) (first new))
                      *pending-issues*))
          (print "the attribute name is not valid."))))
 ;; remove any entries for deleted attribute names
 (if (not (eql old '*add))
    (setf *pending-issues*
          (remove (list 'check-attr-val (name c) (name s) (first old))
                  *pending-issues* :test #'equalp))))

;;______________________________________________________________________________
;; Determine if any entries related to invalid input parameter names used in
;; the value of the attr-val structure can be removed.  The pending-iisues list
;; is searched for any entries that have parameter names that are part of the changed
;; or deleted value.  If this parameter name is not in any other value of the atts
;; of the post, and if it is not part of the new value, then that entry in
;; pending-issues can be deleted.
;;   The old value must have been changed to the new value prior to entering this
;; procedure.
;;______________________________________________________________________________
(defun check-post-atts-missing-input-para (c s old new)
 (dolist (one-entry *pending-issues*)
          (if (eql (first one-entry) ':service)
             (if (and (eql (second one-entry) (name c))
                      (eql (third one-entry) (name s))
                      (eql (fourth one-entry) 'post))
                   (if (and (para-name-in-av (seventh one-entry) old)
                            (not (para-name-in-atts (seventh one-entry) s))
                            (not (para-name-in-av (seventh one-entry) new)))
                      (setf *pending-issues*
                             (remove one-entry *pending-issues* :test #'equalp)) )))))

;;______________________________________________________________________________
;; Return true if the input parameter name exists in the attr-val list.
;;______________________________________________________________________________
(defun para-name-in-av (name av-list)
 (if (not (eql av-list '*delete))
    (if (listp (second av-list))
          (any-member name (second av-list))
        (any-member name (list (second av-list))))))

;;______________________________________________________________________________
;; Return true if the parameter name exists in any of the value slots of the attr-val
;; lists of the attr slot of the post of a service.
;;______________________________________________________________________________
(defun para-name-in-atts (name s)
 (if (post s)
    (if (postf-atts (post s))
          (any-member name (mapcar #'attr-val-value (postf-atts (post s))))
          '())
    '()))

;;*****************************************************************************
;; Modify the messages slot of the post of a service.
;;   A message can be added, an existing message deleted, or an existing message
;; replaced.  When a message is added or replaced, the new message is checked for
;; validity of the class and the service.  If either is invalid, an entry is added
;; to pending-issues.  When an existing message is deleted or replaced, any entries
;; in pending-issues for the old message are removed.
;;   old-massage is the existing (class service) list, or it is (*add) if a message
;; is being added.
;;   new-message is a new (class service) list, or it is (*delete) if a mesage is
;; being deleted.
;;*****************************************************************************
(defun change-serv-post-mess (class-name service-name old-message new-message)
 (symbolp class-name)
 (symbolp service-name)
 (listp old-message)
```

```
  (listp new-message)
  (let* ((the-class (return-class class-name))
         (the-service (return-service class-name service-name))
         (the-post (post the-service)))
   (cond ((eql (first old-message) '*add)
          (if (eql (length new-message) 2)
             (if (unique-message the-post new-message)
                   (add-to-post-messages the-class the-service new-message)
                   (print "The message already exists in the post."))
             (print "The new message list did not contain two elements.")))
         ((eql (first new-message) '*delete)
          (if (not (unique-message the-post old-message))
             (remove-from-post-messages the-class the-service old-message)
             (print "The old message does not exist.")))
         (t
          (if (eql (length old-message) 2)
             (if (eql (length new-message) 2)
                   (if (not (unique-message the-post old-message))
                      (if (unique-message the-post new-message)
                            (replace-post-message the-class the-service
                                                      old-message new-message)
                            (print "The new message already exists in the post."))
                      (print "The old message does not exists in the post."))
                   (print "The new message did not contain two elements."))
             (print "The old message did not contain two elements."))) )))

;;________________________________________________________________________
;; Return true if there is no message in the messages slot of post that match
;; a-message.
;;________________________________________________________________________
(defun unique-message (a-post a-message)
 (if a-post
    (if (postf-messages a-post)
          (let ((result t))
            (dolist (one-mess (postf-messages a-post) result)
                  (if (equal one-mess a-message)
                        (return '()))))
          t)
    t))

;;________________________________________________________________________
;; Add a new message to the messages slot of the post of a service. Call
;; check-post-atts-mess to examine the validity of the class and service used in
;; the new message.
;;________________________________________________________________________
(defun add-to-post-messages (c s nm)
 (if (post s)
    (setf (postf-messages (post s))
            (cons nm (postf-messages (post s))))
    (setf (post s)
            (make-postf :messages '(nm))))
 (check-post-atts-mess c s '*add nm))

;;________________________________________________________________________
;; Remove an existing message from the messages slot of the post of a service.
;; Call check-post-atts-mess to remove an entries in pending-issues for the
;; deleted message.
;;___________________________________________________________________________
(defun remove-from-post-messages (c s om)
 (setf (postf-messages (post s))
        (remove om (postf-messages (post s)) :test #'equal))
 (check-post-atts-mess c s om '*delete))

;;___________________________________________________________________________
;; Replace an existing message in the messages slot of the post of a service with a
;; new message. Call check-post-atts-mess to check the validity of the class and
;; service of the new message and remove entries in pending-issues for the old
;; message.
;;___________________________________________________________________________
(defun replace-post-message (c s om nm)
 (setf (postf-messages (post s))
        (substitute nm om (postf-messages (post s)) :test #'equal))
 (check-post-atts-mess c s om nm))

;;___________________________________________________________________________
;; Check new message for the validity of the class and service. If either is not
;; valid, add an entry to pending issues. Also remove any entries in pending-issues
;; for deleted or replaced messages.
;;___________________________________________________________________________
(defun check-post-atts-mess (c s om nm)
 (if (not (eql nm '*delete))
    (if (check-messages (list nm))
          (progn
            (setf *pending-issues*
                  (cons (list 'check-messages (name c) (name s) nm)
                        *pending-issues*))
            (print "The class and/or service of the message are not valid."))))
 (if (not (eql om '*add))
    (setf *pending-issues*
          (remove (list 'check-messages (name c) (name s) om)
                  *pending-issues* :test #'equalp))))

;;*****************************************************************************
;; Delete a service.
;; The service of a parent cannot be deleted.
;; Any pending-issues for the service are removed.
;; Pending-issues are added for any messages in a service that refer to the deleted
;; service.
;;*****************************************************************************
(defun delete-service (class-name service-name)
 (symbolp class-name)
 (symbolp service-name)
 (let ((the-class (return-class class-name)))
  (if (member service-name (ser-names the-class))
        (if (not (member service-name (third (assoc class-name *necessary-classes*))))
            (let ((the-service (return-service class-name service-name)))
              (if (not (parentp the-class))
                    (progn
                      (setf (services the-class)
                            (remove the-service (services the-class) :test #'equalp))
                      ;; remove pending issues for the service
                      (setf *pending-issues*
                    (remove-serv-entries class-name service-name *pending-issues*))
               ;; add pending-issues on that service
               (dolist (one-class *list-of-classes*)
                    (serv-del-check one-class class-name service-name)))
              (print "The class is a parent, therefore the service cannot be deleted.")))
         (print "The service cannot be deleted."))
     (print "The service is not in the class."))))

;;_________________________________________________________________________
;; Add entries to pending-issues for messages that contain cn and sn.
;;_________________________________________________________________________
(defun serv-del-check (c cn sn)
 (dolist (one-serv (services c))
      (if (post one-serv)
         (dolist (one-mess (postf-messages (post one-serv)))
               (if (and (eql (first one-mess) cn)
                        (eql (second one-mess) sn))
                    (setf *pending-issues*
                         (cons (list 'check-messages (name c) (name one-serv)
                                   one-mess) *pending-issues*))))))
 (setf *pending-issues*
      (remove-duplicates *pending-issues* :test #'equalp)))

;;_________________________________________________________________________
;; Remove entries in pending issues for a deleted services in a class.
;;_________________________________________________________________________
(defun remove-serv-entries (cn sn pending)
 (cond ((null pending) '())
      ((or (eql (first (first pending)) ':service)
         (eql (first (first pending)) 'check-parameter)
         (eql (first (first pending)) 'check-attr-val)
         (eql (first (first pending)) 'check-messages))
       (if (and (eql (second (first pending)) cn)
               (eql (third (first pending)) sn))
         (remove-serv-entries cn sn (rest pending))
         (cons (first pending) (remove-serv-entries cn sn (rest pending)))))
      (t (cons (first pending) (remove-serv-entries cn sn (rest pending))))))

;;***********************************************************************************
;; Add a service without using a service template.
;;  This creates a new blank template for a service consisting of just the name and
;; the description.  Entries in pending-issues involving the new service name are removed.
;; Then the user will enter parameter for the input set and they will be entered using
;; "add-to-io-set"; parameters for the output-set are added using "add-to-io-set";
;; the pre is added using "change-serv-pre"; the atts portion of the post is added using
;; "change-serv-post-atts"; the messages portion of the post is added using
;; "change-serv-post-mess."
;;***********************************************************************************
(defun add-service (class-name service-name desc)
 (symbolp class-name)
 (symbolp service-name)
 (stringp desc)
 (let ((the-class (return-class class-name)))
  (if (not (member service-name (ser-names the-class)))
      (progn
       (setf (services the-class)
                (cons (make-instance 'service
                                     :name service-name
                                     :desc desc
                                     :input-set '()
                                     :output-set '()
                                     :pre '()
                                     :post '()
                                     :verif t)
                      (services the-class)))
          (setf *pending-issues*
                (remove-missing-serv-entries *pending-issues*)))
      (print "The service name is already the name of a service in the class."))))

;;____________________________________________________________________________________
;; Remove any entries in *pending-issues* for invalid messages that are now valid.
;;____________________________________________________________________________________
(defun remove-missing-serv-entries (pending)
  (cond ((null pending) '())
        ((eql (first (first pending)) 'check-messages)
         (if (check-messages (list (fourth (first pending))))
           (cons (first pending) (remove-missing-serv-entries (rest pending)))
           (remove-missing-serv-entries (rest pending))))
        (t (cons (first pending) (remove-missing-serv-entries (rest pending))))))

;;************************************************************************************
;; Add a service based on one of the four service templates:
;;   1. change the value of an attribute
;;   2. return the value of an attribute
;;   3. add a value to an attribute that is a list of values.
;;   4. remove a value from an attribute that is a list of values.
;; Remove any entries in pending issues for invalid messages that are now valid.
;;************************************************************************************
(defun add-template (class-name template-name attribute-name ser-name
                                 &optional (message-list '()))
  (symbolp class-name)
  (symbolp template-name)
  (symbolp attribute-name)
  (if (member attribute-name (att-names (return-class class-name)))
    (if (member template-name '(change return add remove))
        (let* ((the-class (return-class class-name))
               (the-att (return-attribute class-name attribute-name))
               (new-serv (cond ((eql template-name 'change)
                                (change-att-template the-att message-list))
                               ((eql template-name 'return)
                                (return-att-template the-att))
                               ((eql template-name 'add)
                                (if (listp (attrs-base (a-set the-att)))
                                  (add-element-template the-att)
                                  '()))
                               ((eql template-name 'remove)
                                (if (listp (attrs-base (a-set the-att)))
                                  (add-element-template the-att)
                                  '())) )))
          (if new-serv
              (progn
               (setf (name new-serv) ser-name)
```

```
                (setf (verif new-serv) t)
                (if (not (member (name new-serv) (ser-names the-class)))
                   (progn
                       (setf (services the-class)
                           (cons new-serv (services the-class)))
                       (setf *pending-issues*
                           (remove-missing-serv-entries *pending-issues*)))
                   (print "The service already exists in the class.")))
             (print "The attribute is not a list.")))
         (print "The template name is not valid."))
    (print "The attribute is not in the class.")))

;;****************************************************************************
;; Change a whole-part or other relation in a class.
;;  The relations can change in one of five ways:
;;  1. The ranges on the relation can change.  This requires the relation exists and
;; the ranges are a list of two integers.
;;  2.  A relation can be added.  An existing class and a new relation is given.  If
;; the other class in the relation exists, the relation is added to the other class
;; as well.  If the other class does not exist, an entry is added to pending issues
;; stating there is a missing class and a relation that belongs to that class.
;;  3.  A relation can be deleted.  The relation is deleted in the other class, if
;; the other class exists.  If the other class does not exists, the entry in
;; pending issues stating there is a missing class and relation is deleted.
;;  4.  The other class of a relation can be changed.  The relation must be removed
;; from the other class, if the other class exists.  If it does not exist, the entry in
;; pending issues on the missing class and relation is deleted.  If the new class
;; exists, the new relation is added to it.  Otherwise, an entry is added to
;; pending issues stating there is a missing class and a relation that belongs to
;; that class.
;;  5.  The name of a relation can change.  The relation cannot be a whole/part
;; relation.  The name must be changed in the class and the other class, if it exists.
;; If it does not exist, there must be an entry in pending-issues.  This entry must
;; be changed to reflect the new name of the relation.
;;****************************************************************************

;;______________________________________________________________________________
;; Change the ranges of an existing whole/part or other relation.
;; Input parameters:
;;   class-name : the class name where the relation structure exists.
;;   old-relation : the existing relation structure.
;;   new-range1 : the new range for the class1 slot of the relation.  A list of
;; two elements.
;;   new-range2 : the new range for the class2 slot of the relation.  A list of
;; two elements.
;;    The ranges are changed if the relation exists in class-name and the new ranges
;; are lists of two integers.  If the other class in the relation exists (the other
;; class is the class in the relation other than class-name), the ranges are changed
;; in its relation also.  If not, the pending-issues entry on the missing class and
;; relation is deleted, and a new one is added for the new relation.
;;______________________________________________________________________________
(defun change-relation-range (class-name old-relation new-range1 new-range2)
  (symbolp class-name)
  (typep old-relation 'relation)
  (listp new-range1)
  (listp new-range2)
```

```
(let* ((the-class (return-class class-name))
       (other-class-name (if (eql (relation-class1 old-relation) class-name)
                             (relation-class2 old-relation)
                             (relation-class1 old-relation)))
       (other-class (if (member other-class-name (class-namel))
                        (return-class other-class-name)
                        '())))
  (if (and (eql (length new-range1) 2)
           (eql (length new-range2) 2))
      (let ((new-relation (make-relation :name (relation-name old-relation)
                                         :class1 (relation-class1 old-relation)
                                         :range1 new-range1
                                         :class2 (relation-class2 old-relation)
                                         :range2 new-range2)))
        (if (eql (relation-name old-relation) 'whole/part)
            (if (member old-relation (whole-part the-class) :test #'equalp)
                (progn
                  (setf (whole-part the-class)
                        (substitute new-relation old-relation
                                    (whole-part the-class) :test #'equalp))
                  (if other-class
                      (setf (whole-part other-class)
                            (substitute new-relation old-relation
                                        (whole-part other-class) :test #'equalp))
                      (progn
                        ;; remove the existing entry and add a new one for the
                        ;; new relation
                        (remove-relation-class-missing other-class-name old-relation)
                        (add-relation-class-missing other-class-name new-relation))))
                (print "The old relation is not in the class."))
            ;; an other relation
            (if (member old-relation (relation the-class) :test #'equalp)
                (progn
                  (setf (relation the-class)
                        (substitute new-relation old-relation
                                    (relation the-class) :test #'equalp))
                  (if other-class
                      (setf (relation other-class)
                            (substitute new-relation old-relation
                                        (relation other-class) :test #'equalp))
                      (progn
                        (remove-relation-class-missing other-class-name old-relation)
                        (add-relation-class-missing other-class-name new-relation))))
                (print "The old relation is not in the class."))) )
      (print "The new ranges do not contain two elements.")) ))

;;___________________________________________________________________________
;; Add a new relation to an existing class.
;; Input parameters:
;;   class-name : the existing class
;;   new-relation : the relation structure that is to be added.
;; If the other class in the relation exists, the new relation is added to the
;; other class. If not, an entry is added to pending-issues stating the other
;; class and the new relation do not exist.
;;___________________________________________________________________________
(defun add-new-relation (class-name new-relation)
 (symbolp class-name)
 (typep new-relation 'relation)
 ;; check to ensure the new relation has a proper structure
 (if (and (eql (length (relation-range1 new-relation)) 2)
          (eql (length (relation-range2 new-relation)) 2)
          (or (eql (relation-class1 new-relation) class-name)
             (eql (relation-class2 new-relation) class-name))
          (not (eql (relation-class1 new-relation) (relation-class2 new-relation))))
   (let* ((the-class (return-class class-name))
          (other-class-name (if (eql (relation-class1 new-relation) class-name)
                                (relation-class2 new-relation)
                                (relation-class1 new-relation)))
          (other-class (if (member other-class-name (class-namel))
                           (return-class other-class-name)
                           '())))
     ;; check to ensure not in class already
     (if (eql (relation-name new-relation) 'whole/part)
       (if (not (member new-relation (whole-part the-class) :test #'equalp))
           (progn
             (setf (whole-part the-class)
                   (cons new-relation (whole-part the-class)))
             (if other-class
               (setf (whole-part other-class)
                     (cons new-relation (whole-part other-class)))
               (add-relation-class-missing other-class-name new-relation)))
         (print "The relation already exists in the class."))
       (if (not (member new-relation (relation the-class) :test #'equalp))
           (progn
             (setf (relation the-class)
                   (cons new-relation (relation the-class)))
             (if other-class
               (setf (relation other-class)
                     (cons new-relation (relation other-class)))
               (add-relation-class-missing other-class-name new-relation)))
         (print "The relation already exists in the class.")) ))
   (print "The new relation does not have the proper structure.")))

;;___________________________________________________________________________
;; Delete an existing relation from a class
;; Input parameters:
;;   class-name : an existing class that conatins the relation to be removed.
;;   old-relation : the relation to be removed.
;; If the other class in the relation exists, the relation is removed from that
;; class.  Otherwise, the entry in pending issues for that missing class and
;; relation is removed.
;;___________________________________________________________________________
(defun delete-relation (class-name old-relation)
 (symbolp old-relation)
 (typep old-relation 'relation)
 (let* ((the-class (return-class class-name))
        (other-class-name (if (eql (relation-class1 old-relation) class-name)
                              (relation-class2 old-relation)
                              (relation-class1 old-relation)))
        (other-class (if (member other-class-name (class-namel))
                         (return-class other-class-name)
                         '())))
   (if (eql (relation-name old-relation) 'whole/part)
       (if (member old-relation (whole-part the-class) :test #'equalp)
          (progn
            (setf (whole-part the-class)
                   (remove old-relation (whole-part the-class) :test #'equalp))
            (if other-class
                (setf (whole-part other-class)
                       (remove old-relation (whole-part other-class)
                                  :test #'equalp))
                (remove-relation-class-missing other-class-name old-relation)))
          (print "The relation is not in the whole-part structure."))
     (if (member old-relation (relation the-class) :test #'equalp)
          (progn
            (setf (relation the-class)
                   (remove old-relation (relation the-class) :test #'equalp))
            (if other-class
                (setf (relation other-class)
                       (remove old-relation (relation other-class)
                                  :test #'equalp))
                (remove-relation-class-missing other-class-name old-relation)))
          (print "The relation is not in the relation structure.")))))

;;______________________________________________________________________________
;; Change the other class name of a relation
;; Input parameters:
;;   class-name : an existing class with the relation to be changed
;;   old-relation : the relation to be changed in class-name
;;   new-class-name : the new class name for the other class name in old-relation.
;; The other class name is the class name in the relation that is not class-name.
;;______________________________________________________________________________
(defun change-relation-class (class-name old-relation new-class-name)
 (symbolp class-name)
 (typep old-relation 'relation)
 (symbolp new-class-name)
 (let* ((the-class (return-class class-name))
        (other-class-name (if (eql (relation-class1 old-relation) class-name)
                                 (relation-class2 old-relation)
                               (relation-class1 old-relation)))
        (other-class (if (member other-class-name (class-namel))
                           (return-class other-class-name)
                           '()))
        (other-new-class (if (member new-class-name (class-namel))
                           (return-class new-class-name)
                           '()))
        (new-relation (make-relation :name (relation-name old-relation)
                                  :class1 (if (eql (relation-class1 old-relation)
                                                    class-name)
                                               class-name
                                               new-class-name)
                                  :range1 (relation-range1 old-relation)
                                  :class2 (if (eql (relation-class2 old-relation)
                                                    class-name)
                                               class-name
                                               new-class-name)
                                  :range2 (relation-range2 old-relation))))
   (if (eql (relation-name old-relation) 'whole/part)
      (if (member old-relation (whole-part the-class) :test #'equalp)
        (progn
         (setf (whole-part the-class)
               (substitute new-relation old-relation
                           (whole-part the-class) :test #'equalp))
         (if other-class
             (setf (whole-part other-class)
                   (remove old-relation (whole-part other-class)
                           :test #'equalp))
             (remove-relation-class-missing other-class-name old-relation))
         (if other-new-class
             (setf (whole-part other-new-class)
                   (cons new-relation (whole-part other-new-class)))
             (add-relation-class-missing new-class-name new-relation)) )
        (print "The old relation is not part of the whole-part structure."))
    (if (member old-relation (relation the-class) :test #'equalp)
        (progn
         (setf (relation the-class)
               (substitute new-relation old-relation
                           (relation the-class) :test #'equalp))
         (if other-class
             (setf (relation other-class)
                   (remove old-relation (relation other-class)
                           :test #'equalp))
             (remove-relation-class-missing other-class-name old-relation))
         (if other-new-class
             (setf (relation other-new-class)
                   (cons new-relation (relation other-new-class)))
             (add-relation-class-missing new-class-name new-relation)) )
        (print "The old relation is not part of the relation structure.")) )))

;;___________________________________________________________________________
;; Change the name of a relation.
;;  The relation can only be an other relation, not a whole-part relation.
;; Input parameters:
;;   class-name : This class must exist
;;   old-relation : a relation that exists in class-name
;;   new-relation-name : the new name for old-relation
;;___________________________________________________________________________
(defun change-relation-name (class-name old-relation new-relation-name)
 (symbolp class-name)
 (typep old-relation 'relation)
 (symbolp new-relation-name)
 (if (not (eql (relation-name old-relation) 'whole-part))
   (let* ((the-class (return-class class-name))
          (other-class-name (if (eql (relation-class1 old-relation) class-name)
                                (relation-class2 old-relation)
                                (relation-class1 old-relation)))
          (other-class (if (member other-class-name (class-namel))
                           (return-class other-class-name)
                           '()))
          (new-relation
           (make-relation :name new-relation-name
                          :class1 (relation-class1 old-relation)
                          :range1 (relation-range1 old-relation)
                          :class2 (relation-class2 old-relation)
```

```
                              :range2 (relation-range2 old-relation))))
          (if (member old-relation (relation the-class) :test #'equalp)
             (progn
               (setf (relation the-class)
                     (substitute new-relation old-relation
                                 (relation the-class) :test #'equalp))
               (if other-class
                   (setf (relation other-class)
                         (substitute new-relation old-relation
                                     (relation other-class) :test #'equalp))
                   (progn
                     ;; remove the existing entry and a new one for the new
                     ;; relation
                     (remove-relation-class-missing other-class-name old-relation)
                     (add-relation-class-missing other-class-name new-relation))))
             (print "The relation is not in the class.")))
        (print "The relation is a whole/part relation - the name can't be changed.")))

;;______________________________________________________________________________
;; Add an entry to pending-issues that a class does not exist and there is a relation
;; that belongs in that class.
;;______________________________________________________________________________
(defun add-relation-class-missing (class-name the-relation)
  (symbolp class-name)
  (typep the-relation 'relation)
  (setf *pending-issues*
        (cons (list 'missing-class-and-relation class-name the-relation)
              *pending-issues*)))

;;______________________________________________________________________________
;; Remove an entry in pending-issues about a missing class and its associated
;; relation.
;;______________________________________________________________________________
(defun remove-relation-class-missing (class-name a-relation)
  (symbolp class-name)
  (typep a-relation 'relation)
  (setf *pending-issues*
        (remove
         (list 'missing-class-and-relation class-name a-relation)
         *pending-issues* :test #'equalp)))

;;*****************************************************************************************
;; Change the inheritance structure.
;;   There are three permissible changes:
;; 1. Remove a parent of a class.  The class must not be the parent of any other class.
;; The attributes used in attributes adn services are checked to ensure they are still
;; valid.  If any are not, entries are added to pending-issues.
;; 2. Add a parent.  The parent must exist in the model.  Pending-issues is checked
;; for any entries that can be removed due to the addition of the new parent.
;; 3. Change a parent.  This involves deleting an existing parent and adding a new one.
;;*****************************************************************************************

;;______________________________________________________________________________
;; Remove a parent of a class.
;;  The class must not be the parent of any other class.  The attributes and
;; services are checked for attributes adn/or services that are not valid as a result.
;; Entries are added to pending-issues for any invalid attributes and services.
;;______________________________________________________________________________
(defun remove-parent (class-name parent-name)
 (symbolp class-name)
 (symbolp parent-name)
 (let ((the-class (return-class class-name)))
   (if (not (parentp the-class))
        (if (member parent-name (inheritance the-class))
           (progn
             (setf (inheritance the-class)
                   (remove parent-name (inheritance the-class) :test #'equal))
             ;; check attributes for invalid attributes used in a-set
             (dolist (one-attr (state-space the-class))
                   (if (atts-attc one-attr)
                         (setf *pending-issues*
                               (cons (list 'atts-attc class-name (name one-attr))
                                   *pending-issues*))))
             ;; check attributes and services used in the services
             (dolist (one-serv (services the-class))
                   (dolist (input-para (input-set one-serv))
                         (if (check-parameter class-name input-para)
                               (setf *pending-issues*
                                     (cons (list 'check-parameter class-name
                                                 (name one-serv) 'input-set
                                                 input-para) *pending-issues*))))
                   (dolist (output-para (output-set one-serv))
                         (if (check-parameter class-name output-para)
                               (setf *pending-issues*
                                     (cons (list 'check-parameter class-name
                                                 (name one-serv) 'output-set
                                                 output-para) *pending-issues*))))
                   (if (post one-serv)
                         (if (postf-atts (post one-serv))
                            (dolist (one-attr-val (postf-atts (post one-serv)))
                                  (if (check-attr-val one-attr-val class-name)
                                        (setf *pending-issues*
                                              (cons
                                               (list 'check-attr-val class-name
                                                    (name one-serv)
                                                    (attr-val-name one-attr-val)))))))))
             (setf *pending-issues*
                   (remove-duplicates *pending-issues* :test #'equalp)) )
           (print "The parent is not in the class."))
     (print "The class is a parent of another class.")) ))

;;______________________________________________________________________________
;; Add a parent to a class.
;;   Pending-issues is checked for entries that can be removed due to the
;; inheritance of attributes and services from the new parent.
;;______________________________________________________________________________
(defun add-parent (class-name parent-name)
 (if (member parent-name (class-namel))
    (let ((the-class (return-class class-name)))
        (if (not (member parent-name (inheritance the-class)))
           (progn
             (setf (inheritance the-class)
                (cons parent-name (inheritance the-class)))
          ;; check attributes for invalid attributes used in a-set
          (dolist (one-attr (state-space the-class))
                (if (not (atts-attc one-attr))
                      (setf *pending-issues*
                            (remove (list 'atts-attc class-name (name one-attr))
                                  *pending-issues* :test #'equalp))))
          ;; check attributes and services used in the services
          (dolist (one-serv (services the-class))
                (dolist (input-para (input-set one-serv))
                      (if (not (check-parameter class-name input-para))
                            (setf *pending-issues*
                                  (remove (list 'check-parameter class-name
                                                (name one-serv) 'input-set
                                                input-para) *pending-issues*
                                                :test #'equalp))))
                (dolist (output-para (output-set one-serv))
                      (if (not (check-parameter class-name output-para))
                            (setf *pending-issues*
                                  (remove (list 'check-parameter class-name
                                                (name one-serv) 'output-set
                                                output-para) *pending-issues*
                                                :test #'equalp))))
                (if (post one-serv)
                      (if (postf-atts (post one-serv))
                        (dolist (one-attr-val (postf-atts (post one-serv)))
                              (if (not (check-attr-val one-attr-val class-name))
                                    (setf *pending-issues*
                                          (remove
                                          (list 'check-attr-val class-name
                                                (name one-serv)
                                                (attr-val-name one-attr-val))
                                          *pending-issues* :test #'equalp)))))) ))
          (print "The new parent is already a parent of the class.")))
    (print "The new parent is not in the model.")))

;;________________________________________________________________________
;; Change the parent of a class.
;;   This is two steps - first delete the old parent.  If this step is successful,
;; the new parent is added.
;;________________________________________________________________________
(defun change-parent (class-name old-parent new-parent)
  (let ((the-class (return-class class-name)))
   (if (member old-parent (inheritance the-class))
        (if (member new-parent (class-namel))
          (progn
            (remove-parent class-name old-parent)
            (if (not (member old-parent (inheritance the-class)))
                (add-parent class-name new-parent)))
          (print "The new parent is not in the model."))
      (print "The parent to be replaced is not a parent of the class."))))

;;*********************************************  *************************************
;;
;; Add a class.
;;   The input are a class name and a description.  The remainder of the class's slots
;; are set to blank and filled in by using the following routines:
```

```
;; 1. add-attribute
;; 2. add-service or add-template
;; 3. add-parent
;; 4. add-new-relation
;;  Pending issues resolved by the new class name are removed.  Also, if there are
;; relations for that class name in pending issues, they are added to the class and
;; the pending-issues entries removed.
;;***********************************************************************************
;;
(defun add-class (class-name desc)
 (symbolp class-name)
 (stringp desc)
 (if (not (member class-name (class-namel)))
    (progn
        (setf *list-of-classes*
              (cons (make-instance 'generic-class
                                          :name class-name
                                          :desc desc
                                          :state-space '()
                                          :services '()
                                          :inheritance '()
                                          :whole-part '()
                                          :relation '()
                                          :verif t) *list-of-classes*))
        ;; remove pending issues on a missing class name
        (setf *pending-issues* (new-class-pending *pending-issues*))
        ;; add any relations that are waiting for the new class
        (add-rel-to-new-class class-name))
    (print "The class name is the name of an existing class.")))

(defun new-class-pending (pending)
 (cond ((null pending) '())
        ((eql (first (first pending)) 'atts-classc)
         (if (atts-classc (return-attribute (second (first pending))
                                                    (third (first pending))))
            (cons (first pending) (new-class-pending (rest pending)))
            (new-class-pending (rest pending))))
        ((eql (first (first pending)) 'atts-attc)
         (if (atts-attc (return-attribute (second (first pending))
                                                  (third (first pending))))
            (cons (first pending) (new-class-pending (rest pending)))
            (new-class-pending (rest pending))))
        ((eql (first (first pending)) 'check-parameter)
         (if (remove-null (check-parameter (second (first pending))
                                                  (fifth (first pending))))
            (cons (first pending) (new-class-pending (rest pending)))
            (new-class-pending (rest pending))))
        ((eql (first (first pending)) 'check-messages)
         (if (check-messages (list (fourth (first pending))))
            (cons (first pending) (new-class-pending (rest pending)))
            (new-class-pending (rest pending))))
        (t (cons (first pending) (new-class-pending (rest pending))))))

;;___________________________________________________________________________
;; Add a relation that is waiting for a new class.  The relation is in
;; pending-issues.
;;___________________________________________________________________________
```

```
(defun add-rel-to-new-class (class-name)
 (symbolp class-name)
 (let ((result '()))
  (dolist (one-entry *pending-issues*)
          (if (and (eql (first one-entry) 'missing-class-and-relation)
                   (eql (second one-entry) class-name))
               (setf result (cons one-entry result))))
  (if result
       (dolist (one-rel result)
               (if (eql (relation-name (third one-rel)) 'whole/part)
                  (setf (whole-part (return-class class-name))
                        (cons (third one-rel) (whole-part (return-class class-name))))
                  (setf (relation (return-class class-name))
                        (cons (third one-rel) (relation (return-class class-name)))))
               (setf *pending-issues*
                   (remove one-rel *pending-issues* :test #'equalp)) ))))

;;************************************************************************************
;; Delete a class.
;;  A class can only be deleted if it is not the parent of another class.
;; Deleting a class removes entries in pending-issues for that class and adds
;; entries in pending issues if any components of attributes or services become invalid.
;; All relations associated with that class are deleted.
;;************************************************************************************
(defun delete-class (class-name)
 (symbolp class-name)
 (if (member class-name (class-namel))
   (if (not (assoc class-name *necessary-classes*))
        (let ((the-class (return-class class-name)))
          (if (not (parentp the-class))
               (progn
                (dolist (one-wp (whole-part the-class))
                        (delete-relation class-name one-wp))
                (dolist (one-rel (relation the-class))
                        (delete-relation class-name one-rel))
                (setf *list-of-classes*
                      (remove the-class *list-of-classes* :test #'equalp))
                ;; remove from pending issues
                (setf *pending-issues*
                      (remove-a-class-pending class-name *pending-issues*))
                ;; add to pending issues*
                (dolist (one-class *list-of-classes*)
                        (attr-del-check one-class)
                        (dolist (one-serv (services the-class))
                                (serv-del-check one-class class-name (name one-serv)))))
             (print "The class is a parent.")))
        (print "The class cannot be deleted."))
   (print "The class is not in the model.")))

;; cn is the name of the class being deleted.
(defun remove-a-class-pending (cn pending)
 (symbolp cn)
 (cond ((null pending) '())
       ((and (eql (second (first pending)) cn)
             (not (eql (first (first pending)) 'missing-class-and-relation)))
        (remove-a-class-pending cn (rest pending))
        (cons (first pending) (remove-a-class-pending cn (rest pending))))
       ((eql (first (first pending)) 'missing-class-and-relation)
        (let ((other-class (if (eql (relation-class1 (third (first pending)))
                                    (second (first pending)))
                                (relation-class2 (third (first pending)))
                                (relation-class1 (third (first pending))))))
          (if (eql other-class cn)
            (remove-a-class-pending cn (rest pending))
            (cons (first pending) (remove-a-class-pending cn (rest pending))))))
       (t (cons (first pending) (remove-a-class-pending cn (rest pending))))))

;;*****************************************************************************
;; Model checks that are advisory checks and not absolute checks.
;; These issues are those that should be brought to the user's attention before the
;; model is complete, but are not required to be resolved. They are evaluated before
;; the model is complete or whenever the user requests.
;;*****************************************************************************
(defun advisory-tests ()
 (setf *advisory-issues* '())
 (dolist (one-class *list-of-classes*)
         ;; is the class connected to other classes in the model
         (if (connectionp one-class)
            (remove-advisory (name one-class) 'connectionp)
            (add-advisory (name one-class) 'connectionp))
         ;; if the class is a parent, doe sit have at least two subclasses
         (if (two-subclass-check one-class)
            (add-advisory (name one-class) 'two-subclass-check)
            (remove-advisory (name one-class) 'two-subclass-check))
         ;; if the class has one or less attributes
         (if (one-attributep one-class)
            (add-advisory (name one-class) 'one-attributep)
            (remove-advisory (name one-class) 'one-attributep))
         ;; if the class has one or less services
         (if (one-servicep one-class)
            (add-advisory (name one-class) 'one-servicep)
            (remove-advisory (name one-class) 'one-servicep))
         ;; if the class shares 80% or more of atts and servs with another class
         (if (share-att-serv one-class)
            (add-advisory (name one-class) 'share-att-serv)
            (remove-advisory (name one-class) 'share-att-serv))
         (if (> (class-depth one-class) 2)
            (add-advisory (name one-class) 'class-depth)
            (remove-advisory (name one-class) 'class-depth)))
 (setf *advisory-issues*
         (remove-duplicates *advisory-issues* :test #'equal)))


(defun remove-advisory (class-name test-name)
 (symbolp class-name)
 (symbolp test-name)
 (setf *advisory-issues*
         (remove (list test-name class-name) *advisory-issues* :test #'equal)))

(defun add-advisory (class-name test-name)
 (symbolp class-name)
 (symbolp test-name)
 (setf *advisory-issues*
       (cons (list test-name class-name) *advisory-issues*)))

;;*****************************************************************************
;; Utilities
;;*****************************************************************************


;;_____________________________________________________________________________
;; Replaces all occurances of old by new.  Old and new must be elements.
;;_____________________________________________________________________________
(defun subst* (new old l)
 (check-type l list)
 (cond ((null l) '())
       ((eql old (first l))
        (cons new (subst* new old (rest l))))
       ((listp (first l))
        (cons (subst* new old (first l)) (subst* new old (rest l))))
       (t (cons (first l) (subst* new old (rest l)))) ))

;;_____________________________________________________________________________
;; Return true if elmt is a member of any level of the list l.
;;_____________________________________________________________________________
(defun any-member (elmt l)
 (listp l)
 (cond ((null l) '())
       ((equal (first l) elmt)
        t)
       ((listp (first l))
        (or (any-member elmt (first l))
            (any-member elmt (rest l))))
       (t (any-member elmt (rest l))) ))

;;*****************************************************************************
;; A class cannot have its need-verified slot to true (stating it has been
;; verified) until all the attributes and services of that class have been
;; verfied.  When the user creates a new class, attribute or service, the
;; need-verified is set to true becuase the user must need it or would not
;; have created it.  It is only necessary to check the attributes and services
;; were in the original OAKS domain model.
;;*****************************************************************************
(defun verify-class (class-name)
 (symbolp class-name)
 (let ((the-class (return-class class-name)))
  (if (and (eval (cons 'and (mapcar #'verif (state-space the-class))))
           (eval (cons 'and (mapcar #'verif (services the-class)))))
      (progn
        (setf (verif the-class) t)
        (if (and (mapcar #'verif *list-of-classes*))
            (remove '(classes need verified) *pending-issues* :test #'equalp)))
   (print "All attributes and services have not been verified"))))
```

OAKSAVE.LISP

Code to Save Model Changes

```
(in-package 'oaks)

;;*****************************************************************************
;; This file contains two procedures: write-data and read-data.
;; Write-data writes all the classes in *list-of-classes* and *pending-issues*
;; to a file named userfil.
;; Read-data reads all the information from userfil and creates *list-of-classes*
;; and *pending-issues*
;;*****************************************************************************

;;****************************************************************************
;; Write-data
;;****************************************************************************

(defun write-data ()
  (with-open-file (ofile "userfil" :direction :output)
          (write (length *list-of-classes*) :stream ofile)
          (dolist (one-class *list-of-classes*)
                    (write
                     (list
                      (name one-class)
                      (desc one-class)
                      (save-state-space (state-space one-class))
                      (save-services (services one-class))
                      (inheritance one-class)
                      (save-relations (whole-part one-class))
                      (save-relations (relation one-class))
                      (verif one-class))
                     :stream ofile))
          (write *pending-issues* :stream ofile)))

(defun save-state-space (list-of-atts)
  (let ((result '()))
   (if (null list-of-atts) '()
    (dolist (one-att list-of-atts)
            (setf result
                    (append result
                              (list
                               (list (name one-att)
                                      (desc one-att)
                                      (initial-value one-att)
                                      (if (listp (attrs-base (a-set one-att)))
                                        (let ((bases '()))
                                         (dolist (one-base (attrs-base (a-set one-att)) bases)
                                                 (setf bases
                                                        (append
                                                         bases
                                                         (list
                                                          (list
                                                           (attrs-base one-base)
                                                           (attrs-lower one-base)
                                                           (attrs-upper one-base)))))))
                                       (list
                                        (attrs-base (a-set one-att))
                                        (attrs-lower (a-set one-att))
                                        (attrs-upper (a-set one-att))))
                                        (verif one-att))))))
   result))

(defun save-services (list-of-servs)
 (let ((result '()))
  (if (null list-of-servs) '()
   (dolist (one-serv list-of-servs result)
        (setf result
              (append
               result
               (list
                (list (name one-serv)
                      (desc one-serv)
                      (save-input-set (input-set one-serv))
                      (save-output-set (output-set one-serv))
                      (pre one-serv)
                      (save-post-atts (post one-serv))
                      (save-post-mess (post one-serv))
                      (verif one-serv)))))))))

(defun save-input-set (input-set)
 (if (null input-set) '()
  (cons (list (parameterf-name (first input-set))
              (parameterf-values (first input-set)))
        (save-input-set (rest input-set)))))

(defun save-output-set (output-set)
 (if (null output-set) '()
  (cons (parameterf-values (first output-set))
        (save-output-set (rest output-set)))))

(defun save-post-atts (the-post)
 (if (null the-post) '()
  (if (null (postf-atts the-post)) '()
   (save-atts-list (postf-atts the-post)))))

(defun save-atts-list (post-atts)
 (if (null post-atts) '()
  (cons (list (attr-val-name (first post-atts))
              (attr-val-value (first post-atts)))
        (save-atts-list (rest post-atts)))))

(defun save-post-mess (the-post)
 (if (null the-post) '()
  (if (null (postf-messages the-post)) '()
   (postf-messages the-post))))

(defun save-relations (relation-list)
 (if (null relation-list) '()
  (cons (list (relation-name (first relation-list))
              (relation-class1 (first relation-list))
              (relation-range1 (first relation-list))
              (relation-class2 (first relation-list))
              (relation-range2 (first relation-list)))
        (save-relations (rest relation-list)))))
```

```
;;*****************************************************************************
;; Read-data
;;*****************************************************************************

(defun read-data ()
 (with-open-file (ifile "userfil" :direction :input)
     (let ((result '())
            (one-class '()))
          (setf number (read ifile))
          (dotimes (i number)
                    (setf one-class (read ifile))
                    (setf result
                          (cons (make-instance 'generic-class
                                           :name (first one-class)
                                           :desc (second one-class)
                                           :state-space (get-atts (third one-class))
                                           :services (get-servs (fourth one-class))
                                           :inheritance (fifth one-class)
                                           :whole-part (get-relation (sixth one-class))
                                           :relation (get-relation (seventh one-class))
                                           :verif (eighth one-class))
                                result)))
          (setf *list-of-classes* result)
          (setf *pending-issues* (read ifile)))))

(defun get-atts (att-list)
 (let ((result '()))
   (dolist (one-att att-list result)
            (setf result
                    (append
                     result
                     (list
                      (make-instance 'attribute
                                      :name (first one-att)
                                      :desc (second one-att)
                                      :initial-value (third one-att)
                                      :a-set (return-a-set (fourth one-att))
                                      :verif (fifth one-att)))))))

(defun return-a-set (a-set-values)
 (let ((result '()))
   (if (listp (first a-set-values))
         (progn
          (dolist (one-a-set a-set-values)
                    (setf result
                          (append result
                                 (list (make-attrs :base (first one-a-set)
                                                   :lower (second one-a-set)
                                                   :upper (third one-a-set))))))
          (setf result (make-attrs :base result)))
     (setf result (make-attrs :base (first a-set-values)
                              :lower (second a-set-values)
                              :upper (third a-set-values))))))

(defun get-servs (serv-list)
 (if (null serv-list) '()
   (cons (make-instance 'service
                                   :name (first (first serv-list))
                                   :desc (second (first serv-list))
                                   :input-set (get-input-set-in (third (first serv-list)))
                                   :output-set (get-output-set-in (fourth (first serv-list)))
                                   :pre (fifth (first serv-list))
                                   :post (if (and (null (sixth (first serv-list)))
                                                  (null (seventh (first serv-list))))
                                             '()
                                           (make-postf :atts (get-post-atts-in
                                                               (sixth (first serv-list)))
                                                       :messages (seventh (first serv-list))))
                                   :verif (eighth (first serv-list)))
         (get-servs (rest serv-list)))))

(defun get-input-set-in (one-input-set)
  (if (null one-input-set) '()
   (cons (make-parameterf :name (first (first one-input-set))
                                   :values (second (first one-input-set)))
         (get-input-set-in (rest one-input-set)))))

(defun get-output-set-in (one-output-set)
  (if (null one-output-set) '()
   (cons (make-parameterf :values (first one-output-set))
         (get-output-set-in (rest one-output-set)))))

(defun get-post-atts-in (post-atts)
  (if (null post-atts) '()
   (cons (make-attr-val :name (first (first post-atts))
                                   :value (second (first post-atts)))
         (get-post-atts-in (rest post-atts)))))


(defun get-relation (relation-list)
  (if (null relation-list) '()
   (cons (make-relation :name (first (first relation-list))
                                   :class1 (second (first relation-list))
                                   :range1 (third (first relation-list))
                                   :class2 (fourth (first relation-list))
                                   :range2 (fifth (first relation-list)))
         (get-relation (rest relation-list)))))
```

OAKSUI.LISP

LISPView User Interface Code

```
(in-package 'oaks)

;;******************************************************************************
;; Make the top level base window and a panel.
;;******************************************************************************

;;_____________________________________________________________________________
;; Make a top-level base window
;; The top level base window contains a panel and windows.  Each window is
;; has the top-level base window as its parent.  Each menu and button has the
;; panel as its parent.
;;_____________________________________________________________________________
(setf *oaks-window* (make-instance 'LV:base-window
                                   :width 1100
                                   :height 600
                                   :left 0
                                   :top 0
                                   :mapped t
                                   :label "OAKS"
                                   :left-footer "Model View"))

;;_____________________________________________________________________________
;; The panel is the parent for all menus and buttons
;;_____________________________________________________________________________
(let ((br (LV:bounding-region *oaks-window*)))
  (setf *oaks-panel* (make-instance 'LV:panel
                                    :parent *oaks-window*
                                    :left (LV:region-left br)
                                    :top (LV:region-top br)
                                    :width (LV:region-width br)
                                    :height (truncate (LV:region-height br) 17))))

;;******************************************************************************
;; Make the window that contains either all the class names or the components of a
;; particular class.
;;******************************************************************************

(setf *class-select* '())

;;_____________________________________________________________________________
;; The initial value is null, signifying the entire class is selected.
;; The values "name", "desc", "w-p", "rel", and "inh" are used.  If an
;; attribute or service is selected, the value is a list consisting of
;; two elements.  The first is the symbol 'service or 'attribute, and
;; the second is the string representing the name of the attribute or
;; service chosen.
;;_____________________________________________________________________________
(setf *component-select* '())

;;_____________________________________________________________________________
;; Create the class list or the class components that goes in the class-list-vp
;;_____________________________________________________________________________
(defun create-class-list ()
  (let ((beg-x 20)
        (beg-y 30))
    (LV:clear class-list-vp)
```

```
(if *class-select*
    ;; print out all the components of a class
    (let ((the-class (get-class-from-name)))
      (LV:draw-string class-list-vp beg-x beg-y "Class Name:")
      (LV:draw-string class-list-vp (+ 10 beg-x) (+ 15 beg-y)
                      (symbol-name (name the-class)))
      (LV:draw-string class-list-vp beg-x (+ 30 beg-y) "Class-Description:")
      (setf beg-y (print-restricted-width 40 (desc the-class)
                                          class-list-vp (+ 10 beg-x) (+ 45 beg-y)))
      (LV:draw-string class-list-vp beg-x beg-y "Attributes:")
      (setf beg-y (+ 15 beg-y))
      (dolist (one-att (state-space the-class))
              (LV:draw-string class-list-vp (+ 10 beg-x) beg-y
                              (symbol-name (name one-att)))
              (setf beg-y (+ 15 beg-y)))
      (LV:draw-string class-list-vp beg-x beg-y "Services:")
      (setf beg-y (+ 15 beg-y))
      (dolist (one-serv (services the-class))
              (LV:draw-string class-list-vp (+ 10 beg-x) beg-y
                              (symbol-name (name one-serv)))
              (setf beg-y (+ 15 beg-y)))
      (LV:draw-string class-list-vp beg-x beg-y "Whole/Part:")
      (setf beg-y (+ 15 beg-y))
      (dolist (one-wp (whole-part the-class))
              (LV:draw-string class-list-vp (+ 10 beg-x) beg-y "Class1:")
              (LV:draw-string class-list-vp (+ 55 beg-x) beg-y
                              (symbol-name (relation-class1 one-wp)))
              (LV:draw-string class-list-vp (+ 10 beg-x) (+ 15 beg-y) "Range1:")
              (LV:draw-string class-list-vp (+ 55 beg-x) (+ 15 beg-y)
                              (princ-to-string (relation-range1 one-wp)))
              (LV:draw-string class-list-vp (+ 10 beg-x) (+ 30 beg-y) "Class2:")
              (LV:draw-string class-list-vp (+ 55 beg-x) (+ 30 beg-y)
                              (symbol-name (relation-class2 one-wp)))
              (LV:draw-string class-list-vp (+ 10 beg-x) (+ 45 beg-y) "Range2:")
              (LV:draw-string class-list-vp (+ 55 beg-x) (+ 45 beg-y)
                              (princ-to-string (relation-range2 one-wp)))
              (setf beg-y (+ 60 beg-y)))
      (LV:draw-string class-list-vp beg-x beg-y "Relation:")
      (setf beg-y (+ 15 beg-y))
      (dolist (one-rel (relation the-class))
              (LV:draw-string class-list-vp (+ 10 beg-x) beg-y "Name:")
              (LV:draw-string class-list-vp (+ 55 beg-x) beg-y
                              (symbol-name (relation-name one-rel)))
              (LV:draw-string class-list-vp (+ 10 beg-x) (+ 15 beg-y) "Class1:")
              (LV:draw-string class-list-vp (+ 55 beg-x) (+ 15 beg-y)
                              (symbol-name (relation-class1 one-rel)))
              (LV:draw-string class-list-vp (+ 10 beg-x) (+ 30 beg-y) "Range1:")
              (LV:draw-string class-list-vp (+ 55 beg-x) (+ 30 beg-y)
                              (princ-to-string (relation-range1 one-rel)))
              (LV:draw-string class-list-vp (+ 10 beg-x) (+ 45 beg-y) "Class2:")
              (LV:draw-string class-list-vp (+ 55 beg-x) (+ 45 beg-y)
                              (symbol-name (relation-class2 one-rel)))
              (LV:draw-string class-list-vp (+ 10 beg-x) (+ 60 beg-y) "Range2:")
              (LV:draw-string class-list-vp (+ 55 beg-x) (+ 60 beg-y)
                              (princ-to-string (relation-range2 one-rel)))
              (setf beg-y (+ 75 beg-y)))
```

```
        (LV:draw-string class-list-vp beg-x beg-y "Parents:")
        (setf beg-y (+ 15 beg-y))
        (LV:draw-string class-list-vp (+ 10 beg-x) beg-y
                        (princ-to-string (inheritance the-class)))
        (setf beg-y (+ 15 beg-y))
        (LV:draw-string class-list-vp beg-x beg-y "Verified:")
        (if (verif the-class)
          (LV:draw-string class-list-vp (+ 55 beg-x) beg-y "Yes")
         (LV:draw-string class-list-vp (+ 55 beg-x) beg-y "No")))

   (dolist (one-class *list-of-classes*)
          (LV:draw-string class-list-vp beg-x beg-y
                           (symbol-name (name one-class)))
          (setf beg-y (+ 15 beg-y))
          (if (inheritance one-class)
               (dolist (one-parent (inheritance one-class))
                      (LV:draw-string class-list-vp (+ 10 beg-x) beg-y
                                      (symbol-name one-parent))
                      (setf beg-y (+ 15 beg-y))))) )))

;;______________________________________________________________________
;; The *list-window* is a child of *oaks-window*. It contains a scrolling
;; window.
;;______________________________________________________________________
(setf *list-window* (make-instance 'LV:window
                                   :parent *oaks-window*
                                   :width 300
                                   :height
                                   (- (LV:region-height (LV:bounding-region *oaks-window*))
                                      (LV:region-height (LV:bounding-region *oaks-panel*)))
                                   :left
                                   (LV:region-left (LV:bounding-region *oaks-window*))
                                   :top 35
                                   :mapped t
                                   :label "List Window"))

;;______________________________________________________________________
;; The class-list window is a scrolling window within *list-window*.
;; The dimensions of the visible part of the scrolling window are in the
;; view-region. The entire region that can be viewed though scrolling is
;; shown in output-region.
;; There must be room for the scrollbar, which is why the width of the
;; output region is less than the width of *list-window*
;; The repaint function is used when the window is first created or when
;; the user selects "refresh" from the frame menu using the mouse.
;;______________________________________________________________________
(setf class-list
   (make-instance 'LV:scrolling-window
                  :parent *list-window*
                  :mapped t
                  :output-region (LV:make-region
                                  :width
                                  (LV:region-width
                                   (LV:bounding-region *list-window*))
                                  :height (* 4 (LV:region-height
                                              (LV:bounding-region *list-window*))))
```

```
                    :view-region (LV:make-region
                                   :width
                                   (- (LV:region-width
                                       (LV:bounding-region *list-window*))
                                      20)
                                   :height
                                   (LV:region-height
                                    (LV:bounding-region *list-window*)))
                    :vertical-scrollbar (make-instance 'LV:vertical-scrollbar
                                                       :right
                                                       (LV:region-right
                                                        (LV:bounding-region
                                                         *list-window*)))
                    :repaint #'(lambda (LV:viewport &rest ignore)
                                 (create-class-list))))

;;________________________________________________________________________
;; A viewport is what can be written to in the scrolling window.
;;________________________________________________________________________
(setf class-list-vp (car (LV:viewports class-list)))

;; (clear vp) clears the entire output-region. The repaint function only works
;; when the user selects Refresh from the frame menu using the mousewq

;;*************************************************************************************
;; Create the window the contains the current selected component.
;;*************************************************************************************


;;________________________________________________________________________
;; Display the currently selected component in the *current-component-window*.
;;________________________________________________________________________
(defun display-current-component ()
 (LV:clear current-component-vp)
 (cond ((and *class-select* *component-select*)
        ;; a component of a class has been selected
        (cond ((equal *component-select* "name")
               (display-class-name))
              ((equal *component-select* "desc")
               (display-class-desc))
              ((equal *component-select* "w-p")
               (display-whole-part))
              ((equal *component-select* "rel")
               (display-rel))
              ((equal *component-select* "inh")
               (display-inh))
              ((listp *component-select*)
               (if (equal (first *component-select*) 'attribute)
                   (display-attribute)
                 (display-service))) ))))

(defun display-class-name ()
 (let ((x-pos 20)
       (y-pos 20)
       (the-class (get-class-from-name)))
   (LV:draw-string current-component-vp x-pos y-pos "CLASS NAME:")
   (LV:draw-string current-component-vp x-pos (+ 30 y-pos)
                    (symbol-name (name the-class)))))

(defun display-class-desc ()
 (let ((x-pos 20)
       (y-pos 20)
       (the-class (get-class-from-name)))
  (LV:draw-string current-component-vp x-pos y-pos "CLASS DESCRIPTION:")
  (print-restricted-width 95 (desc the-class) current-component-vp
                          x-pos (+ 15 y-pos))))

(defun display-whole-part ()
 (let ((x-pos 20)
       (y-pos 20)
       (the-class (get-class-from-name)))
  (LV:draw-string current-component-vp x-pos y-pos "CLASS WHOLE/PART STRUCTURE:")
  (setf y-pos (+ 15 y-pos))
  (dolist (one-wp (whole-part the-class))
          (LV:draw-string current-component-vp x-pos y-pos
                          (princ-to-string one-wp))
          (setf y-pos (+ 15 y-pos)))))

(defun display-rel ()
 (let ((y-pos 20)
       (x-pos 20)
       (the-class (get-class-from-name)))
  (LV:draw-string current-component-vp x-pos y-pos "CLASS RELATION STRUCTURE:")
  (setf y-pos (+ 15 y-pos))
  (dolist (one-rel (relation the-class))
          (LV:draw-string current-component-vp x-pos y-pos
                          (princ-to-string one-rel))
          (setf y-pos (+ 15 y-pos)))))

(defun display-inh ()
 (let ((y-pos 20)
       (x-pos 20)
       (the-class (get-class-from-name)))
  (LV:draw-string current-component-vp x-pos y-pos "CLASS PARENTS:")
  (setf y-pos (+ 15 y-pos))
  (LV:draw-string current-component-vp x-pos y-pos
                  (princ-to-string (inheritance the-class)))))

(defun display-attribute ()
 (let* ((y-pos 20)
        (x-pos 20)
        (the-class (get-class-from-name))
        (the-attribute (get-attribute-from-name the-class)))
  (LV:draw-string current-component-vp x-pos y-pos "ONE ATTRIBUTE:")
  (setf y-pos (+ 15 y-pos))
  (LV:draw-string current-component-vp x-pos y-pos "Name:")
  (LV:draw-string current-component-vp (+ 65 x-pos) y-pos
                  (symbol-name (name the-attribute)))
  (setf y-pos (+ 15 y-pos))
  (LV:draw-string current-component-vp x-pos y-pos "Description:")
  (setf y-pos
        (print-restricted-width 85 (desc the-attribute) current-component-vp
                                (+ 10 x-pos) (+ 15 y-pos)))
```

```
   (LV:draw-string current-component-vp x-pos y-pos "Initial Value:")
   (LV:draw-string current-component-vp (+ 85 x-pos) y-pos
                   (princ-to-string (initial-value the-attribute)))
   (setf y-pos (+ 15 y-pos))
   (LV:draw-string current-component-vp x-pos y-pos "Legal Values:")
   (setf y-pos (+ 15 y-pos))
   (if (listp (attrs-base (a-set the-attribute)))
       (progn
        (LV:draw-string current-component-vp (+ 10 x-pos) y-pos "Base:")
        (setf y-pos (+ 15 y-pos))
        (dolist (one-element (attrs-base (a-set the-attribute)))
                (display-one-attrs one-element (+ 20 x-pos) y-pos)
                (setf y-pos (+ 45 y-pos))))
    (progn
       (display-one-attrs (a-set the-attribute) (+ 10 x-pos) y-pos)
       (setf y-pos (+ 45 y-pos))))
   (LV:draw-string current-component-vp x-pos y-pos "Verified:")
   (if (verif the-attribute)
       (LV:draw-string current-component-vp (+ 55 x-pos) y-pos "Yes")
    (LV:draw-string current-component-vp (+ 55 x-pos) y-pos "No"))))

(defun display-one-attrs (one-attr x y)
  (LV:draw-string current-component-vp x y "Base:")
  (LV:draw-string current-component-vp (+ 55 x) y (princ-to-string (attrs-base one-attr)))
  (LV:draw-string current-component-vp x (+ 15 y) "Lower:")
  (LV:draw-string current-component-vp (+ 55 x) (+ 15 y)
                  (princ-to-string (attrs-lower one-attr)))
  (LV:draw-string current-component-vp x (+ 30 y) "Upper:")
  (LV:draw-string current-component-vp (+ 55 x) (+ 30 y)
                  (princ-to-string (attrs-upper one-attr))))

(defun display-service ()
  (let* ((y-pos 20)
         (x-pos 20)
         (the-class (get-class-from-name))
         (the-service (get-service-from-name the-class)))
   (LV:draw-string current-component-vp x-pos y-pos "ONE SERVICE:")
   (setf y-pos (+ 30 y-pos))
   (LV:draw-string current-component-vp x-pos y-pos "Name:")
   (LV:draw-string current-component-vp (+ 65 x-pos) y-pos
                   (symbol-name (name the-service)))
   (setf y-pos (+ 15 y-pos))
   (LV:draw-string current-component-vp x-pos y-pos "Description:")
   (setf y-pos
         (print-restricted-width 85 (desc the-service) current-component-vp
                                 (+ 10 x-pos) (+ 15 y-pos)))
   (LV:draw-string current-component-vp x-pos y-pos "Input Parameters:")
   (setf y-pos (+ 15 y-pos))
   (dolist (one-para (input-set the-service))
           (LV:draw-string current-component-vp (+ 10 x-pos) y-pos "name:")
           (LV:draw-string current-component-vp (+ 60 x-pos) y-pos
                           (princ-to-string (parameterf-name one-para)))
           (LV:draw-string current-component-vp (+ 10 x-pos) (+ 15 y-pos) "values:")
           (LV:draw-string current-component-vp (+ 60 x-pos) (+ 15 y-pos)
                           (princ-to-string (parameterf-values one-para)))
           (setf y-pos (+ 30 y-pos)))
  (LV:draw-string current-component-vp x-pos y-pos "Output Parameters:")
  (setf y-pos (+ 15 y-pos))
  (dolist (one-para (output-set the-service))
          (LV:draw-string current-component-vp (+ 10 x-pos) y-pos "values:")
          (LV:draw-string current-component-vp (+ 60 x-pos) y-pos
                          (princ-to-string (parameterf-values one-para)))
          (setf y-pos (+ 15 y-pos)))
  (LV:draw-string current-component-vp x-pos y-pos "Precondition:")
  (LV:draw-string current-component-vp (+ 80 x-pos) y-pos
                  (princ-to-string (pre the-service)))
  (setf y-pos (+ 15 y-pos))
  (LV:draw-string current-component-vp x-pos y-pos "Postcondition - Changed Attributes")
  (setf y-pos (+ 15 y-pos))
  (if (post the-service)
      (dolist (one-attr-val (postf-atts (post the-service)))
              (LV:draw-string current-component-vp (+ 10 x-pos) y-pos "attribute name:")
              (LV:draw-string current-component-vp (+ 120 x-pos) y-pos
                              (princ-to-string (attr-val-name one-attr-val)))
              (LV:draw-string current-component-vp (+ 10 x-pos) (+ 15 y-pos)
                              "attribute value:")
              (LV:draw-string current-component-vp (+ 120 x-pos) (+ 15 y-pos)
                              (princ-to-string (attr-val-value one-attr-val)))
              (setf y-pos (+ 30 y-pos))))
  (LV:draw-string current-component-vp x-pos y-pos "Postcondition - Messages")
  (setf y-pos (+ 15 y-pos))
  (if (post the-service)
      (dolist (one-mess (postf-messages (post the-service)))
              (LV:draw-string current-component-vp (+ 10 x-pos) y-pos
                              (princ-to-string one-mess))
              (setf y-pos (+ 15 y-pos))))
  (LV:draw-string current-component-vp x-pos y-pos "Verified:")
  (if (verif the-service)
      (LV:draw-string current-component-vp (+ 55 x-pos) y-pos "Yes")
   (LV:draw-string current-component-vp (+ 55 x-pos) y-pos "No"))))

;;______________________________________________________________________
;; The *current-component-window* is a child of *oaks-window*.
;; It contains a scrolling window.
;;______________________________________________________________________
(let ((brw (LV:bounding-region *oaks-window*))
      (brp (LV:bounding-region *oaks-panel*)))
  (setf *current-component-window*
        (make-instance 'LV:window
                       :parent *oaks-window*
                       :width 800
                       :height (+ 50 (truncate (- (LV:region-height brw)
                                                  (LV:region-height brp)) 2))
                       :left (LV:region-width (LV:bounding-region *list-window*))
                       :top (LV:region-top (LV:bounding-region *list-window*))
                       :mapped t
                       :label "Selected Component")))

;;______________________________________________________________________
;; The scroll-current-component window is a srolling window within
;; *cur. ent-component-window*.
;;______________________________________________________________________
```

```
(let ((br (LV:bounding-region *current-component-window*)))
  (setf scroll-current-window
        (make-instance 'LV:scrolling-window
                       :parent *current-component-window*
                       :mapped t
                       :output-region (LV:make-region
                                        :width (LV:region-width br)
                                        :height (* 4 (LV:region-height br)))
                       :view-region (LV:make-region
                                      :width (- (LV:region-width br) 20)
                                      :height (LV:region-height br))
                       :vertical-scrollbar (make-instance 'LV:vertical-scrollbar
                                                          :right
                                                          (LV:region-right br))
                       :repaint #'(lambda (LV:viewport &rest ignore)
                                    (display-current-component)) )))

;;____________________________________________________________________________
;; A viewport is where things are written for scroll-current-window
;;____________________________________________________________________________
(setf current-component-vp (car (LV:viewports scroll-current-window)))

;;******************************************************************************
;; Create the window that contains the pending-issues list.  It contains a scrolling
;; window.
;;******************************************************************************

;;____________________________________________________________________________
;; Display the *pending-issues* list in the *pending-issues-window*
;;____________________________________________________________________________
(defun display-pending-issues ()
  (let ((x 20)
        (y 20))
    (LV:clear pending-issues-vp)
    (LV:draw-string pending-issues-vp x y "PENDING ISSUES:")
    (setf y (+ 25 y))
    (dolist (one-entry *pending-issues*)
            (LV:draw-string pending-issues-vp x y (princ-to-string one-entry))
            (setf y (+ 15 y))) ))

;;____________________________________________________________________________
;; The base window for the *pending-issues* list.
;;____________________________________________________________________________
(let ((brc (LV:bounding-region *current-component-window*)))
  (setf *pending-issues-window*
        (make-instance 'LV:window
                       :parent *oaks-window*
                       :width (LV:region-width brc)
                       :height (LV:region-height brc)
                       :left (LV:region-left brc)
                       :top (LV:region-bottom brc)
                       :mapped t
                       :label "Pending Issues")))

;;____________________________________________________________________________
;; The scrolling window within *pending-issues-window*
```

```
;;_______________________________________________________________________
(let ((br (LV:bounding-region *pending-issues-window*)))
 (setf scroll-pending-issues
        (make-instance 'LV:scrolling-window
                    :parent *pending-issues-window*
                    :mapped t
                    :output-region (LV:make-region
                                  :width (LV:region-width br)
                                  :height (* 4 (LV:region-height br)))
                    :view-region (LV:make-region
                                 :width (- (LV:region-width br) 20)
                                 :height (LV:region-height br))
                    :vertical-scrollbar (make-instance 'LV:vertical-scrollbar
                                                      :right
                                                      (LV:region-right br))
                    :repaint #'(lambda (LV:viewport &rest ignore)
                                (display-pending-issues))) ))

;;_______________________________________________________________________
;; The viewport for scroll-pending-issues
;;_______________________________________________________________________
(setf pending-issues-vp (car (LV:viewports scroll-pending-issues)))

;;***********************************************************************************
;;
;; Create the menu that selects either the entire model or one class.  If the
;; entire model is selected, *class-select* is set to null.  If a class is
;; selected, *class-select* contains a string representation of the class name.
;;***********************************************************************************
;;

(defun make-class-menu ()
 (let ((result '()))
  (dolist (one-class *list-of-classes* result)
         (setf result
               (cons
                (list (symbol-name (name one-class))
                      `(lambda ()
                        (setf *class-select* ,(symbol-name (name one-class)))
                        (setf (LV:state component-select) :active)
                        (setf *component-select* '())
                        (create-class-list)
                        (display-current-component)))
                result)))
  (make-instance 'LV:menu :menu-spec result)))

(defun create-class-choices ()
 (cons
  (first
   (lv:choices (make-instance 'LV:menu :menu-spec
                              '(("Entire Model"
                                 (lambda () (setf *class-select* '())
                                   (redraw-all-windows)
                                   (setf (LV:state component-select) :inactive)))))))
  (lv:choices (make-class-menu))))

(setf get-class-choices
    (make-instance 'LV:menu
                    :choices #'create-class-choices))

(setf model-menu (make-instance 'LV:menu-button
                                :parent *oaks-panel*
                                :label "Model/Class"
                                :menu get-class-choices))
;;____________________________________________________________________________
;; Get the class based on the string representation of the class name
;;____________________________________________________________________________
(defun get-class-from-name ()
 (let ((result '()))
  (dolist (one-class *list-of-classes* result)
        (if (equal *class-select* (symbol-name (name one-class)))
            (progn
             (setf result one-class)
             (return result))))))


;;____________________________________________________________________________
;; Get the attribute based on the string representation of the attribute name.
;;____________________________________________________________________________
(defun get-attribute-from-name (c)
 (let ((result '()))
  (dolist (one-att (state-space c))
        (if (equal (second *component-select*) (symbol-name (name one-att)))
            (progn
             (setf result one-att)
             (return result))))))


;;____________________________________________________________________________
;; Get the service based on the string representation of the service name.
;;____________________________________________________________________________
(defun get-service-from-name (c)
 (let ((result '()))
  (dolist (one-serv (services c))
        (if (equal (second *component-select*) (symbol-name (name one-serv)))
            (progn
             (setf result one-serv)
             (return result))))))

;;*****************************************************************************
;;
;; Create the menu that selects the component of a class
;;*****************************************************************************
;;

;;____________________________________________________________________________
;; The attribute submenu consists of the attribute names.
;;____________________________________________________________________________
(defun make-att-submenu ()
 (let ((result '())
      (the-class (get-class-from-name)))
  (dolist (one-att (state-space the-class) result)
        (setf result
             (cons
              (list (symbol-name (name one-att))
                   `(lambda () (setf *component-select*
                                    (list 'attribute
                                          ,(symbol-name (name one-att))))
```

```
                          (setf (LV:state att-components-menu) :active)
                          (setf (LV:state serv-components-menu) :inactive)
                          (display-current-component)))
                 result)))
   (make-instance 'LV:menu :menu-spec result)))

;;____________________________________________________________________________
;; The service submenu consists of the service names.
;;____________________________________________________________________________
(defun make-serv-submenu ()
 (let ((result '())
        (the-class (get-class-from-name)))
   (dolist (one-serv (services the-class) result)
          (setf result
                 (cons
                  (list (symbol-name (name one-serv))
                         `(lambda () (setf *component-select*
                                               (list 'service
                                                      ,(symbol-name (name one-serv))))
                          (setf (LV:state att-components-menu) :inactive)
                          (setf (LV:state serv-components-menu) :active)
                          (display-current-component)))
                 result)))
   (make-instance 'LV:menu :menu-spec result)))

;;____________________________________________________________________________
;; The component menu contains all the components of a class.
;;____________________________________________________________________________
(setf component-menu1
    (make-instance 'LV:menu
                    :menu-spec
                    '(("Entire Class" (lambda () (setf *component-select* '())
                                          (setf (LV:state att-components-menu) :inactive)
                                          (setf (LV:state serv-components-menu) :inactive)
                                          (display-current-component)))
                      ("Class Name" (lambda () (setf *component-select* "name")
                                          (setf (LV:state att-components-menu) :inactive)
                                          (setf (LV:state serv-components-menu) :inactive)
                                          (display-current-component)))
                      ("Class Description" (lambda () (setf *component-select* "desc")
                                          (setf (LV:state att-components-menu) :inactive)
                                          (setf (LV:state serv-components-menu) :inactive)
                                          (display-current-component)))
                      ("Whole-Part" (lambda () (setf *component-select* "w-p")
                                          (setf (LV:state att-components-menu) :inactive)
                                          (setf (LV:state serv-components-menu) :inactive)
                                          (display-current-component)))
                      ("Relations" (lambda () (setf *component-select* "rel")
                                          (setf (LV:state att-components-menu) :inactive)
                                          (setf (LV:state serv-components-menu) :inactive)
                                          (display-current-component)))
                      ("Inheritance" (lambda () (setf *component-select* "inh")
                                          (setf (LV:state att-components-menu) :inactive)
                                          (setf (LV:state serv-components-menu) :inactive)
                                          (display-current-component))) )))
```

```
(defun att-submenu ()
 (make-instance 'LV:submenu-item
                :label "One Attribute"
                :menu (make-att-submenu)))

(defun serv-submenu ()
 (make-instance 'LV:submenu-item
                :label "One Service"
                :menu (make-serv-submenu)))

;;________________________________________________________________________
;; This menu consists of all the components of a class. "att-submenu" and
;; "serv-submenu" are submenus of the names of the attributes and services
;; of the currently selected class.
;;________________________________________________________________________
(defun get-component-choices ()
 (list (first (LV:choices component-menu1))
       (second (LV:choices component-menu1))
       (third (LV:choices component-menu1))
       (att-submenu)
       (serv-submenu)
       (fourth (LV:choices component-menu1))
       (fifth (LV:choices component-menu1))
       (sixth (LV:choices component-menu1))))

;;________________________________________________________________________
;; The submenus for the attributes and services must be regenerated each
;; time the menus pop up because the class may have changed. The procedure
;; "get-component-choices" is executed each time the menu is shown.
;;________________________________________________________________________
(setf component-menu
      (make-instance 'LV:menu
                     :choices #'get-component-choices))

;;________________________________________________________________________
;; When the menu is first created is when the window is first created.
;; At this point, the entire model is selected. Since there is no class
;; selected, the menu button is set to :inactive. When a class is selected,
;; the menu button is set to active.
;;________________________________________________________________________
(setf component-select
   (make-instance 'LV:menu-button
                  :parent *oaks-panel*
                  :label "Component"
                  :state :inactive
                  :menu component-menu))


;;*****************************************************************************
;; Create the menu that selects an attribute component
;;*****************************************************************************

;;________________________________________________________________________
;; *attribute-component* is null if the whole attribute is selected or contains
;; one of the following strings: "name", "desc", "a-set".
;;________________________________________________________________________
```

```
(setf *attribute-component* '())

;;________________________________________________________________________
;; This menu portrays all attribute components.  The menu does not change.
;; The menu is only active when an attribute is selected.
;;________________________________________________________________________
(setf att-components-menu
   (make-instance 'LV:menu-button
                  :parent *oaks-panel*
                  :label "Attribute Components"
                  :state :inactive
                  :menu
                  (make-instance
                   'LV:menu :menu-spec
                   '(("Entire Attribute" (lambda () (setf *attribute-component* '())))
                     ("Name" (lambda () (setf *attribute-component* "name")))
                     ("Description" (lambda () (setf *attribute-component* "desc")))
                     ("Initial Value" (lambda () (setf *attribute-component* "ini")))
                     ("Legal Values"
                      (lambda () (setf *attribute-component* "a-set")))))))

;;************************************************************************
;; Create the menu that selects the service component
;;************************************************************************


;;________________________________________________________________________
;;  *service-component* is null if the whole service is selected or contains
;; one of the following strings: "name", "desc", "input", "output", "pre",
;; "post-atts", "post-mess".
;;________________________________________________________________________
(setf *service-component* '())

;;________________________________________________________________________
;; This menu portrays all the service components.  This menu does not change.
;; The menu is on;y active when a service is selected.
;;________________________________________________________________________
(setf serv-components-menu
   (make-instance 'LV:menu-button
                  :parent *oaks-panel*
                  :label "Service Components"
                  :state :inactive
                  :menu
                  (make-instance
                   'LV:menu :menu-spec
                   '(("Entire Service" (lambda () (setf *service-component* '())))
                     ("Name" (lambda () (setf *service-component* "name")))
                     ("Description" (lambda () (setf *service-component* "desc")))
                     ("Input Set" (lambda () (setf *service-component* "input")))
                     ("Output Set" (lambda () (setf *service-component* "output")))
                     ("Precondition" (lambda () (setf *service-component* "pre")))
                     ("Postcondition Attributes"
                      (lambda () (setf *service-component* "post-atts")))
                     ("Postcondition Messages"
                      (lambda () (setf *service-component* "post-mess"))) ))))

;;************************************************************************
```

```
;; Create the menu that provides the actions possible based on *class-select*,
;; *component-select*, *attribute-component* and *service-component*
;;*****************************************************************************
;;

;;_______________________________________________________________________________
;; The command is representative of the action the user wants to perform
;; on the currently selected component of the model.
;;_______________________________________________________________________________
(setf *command* '())

;;_______________________________________________________________________________
;; The action is the LISP procedures that must execute to perform the
;; action represented in *command*
;;_______________________________________________________________________________
(setf *action* '())

;;_______________________________________________________________________________
;; The following are menu items, each of which represents one action the user
;; can perform.  Each menu item is tied to a drop-down menu selection.
;;_______________________________________________________________________________

(defun menu-add-class ()
 (make-instance 'LV:command-menu-item
                :label "Add a Class"
                :command
                #'(lambda () (setf *command* "add-class")
                   (setf *current-text-fields*
                         (list "Class name" "Class description"))
                   (display-get-values)
                   (setf *action*
                         '(add-class (read-from-string (LV:value text1))
                                    (LV:value text2))) )))

(defun menu-delete-class ()
 (make-instance 'LV:command-menu-item
                :label "Delete the Class"
                :command
                #'(lambda () (setf *command* "delete")
                   (delete-class (read-from-string *class-select*))
                   (setf *class-select* '())
                   (redraw-all-windows)) ))

(defun menu-change-class-name ()
 (make-instance 'LV:command-menu-item
                :label "Change Class Name"
                :command
                #'(lambda () (setf *command* "change-name")
                   (let ((the-class *class-select*))
                    (setf *class-select* '())
                    (setf *component-select* '())
                    (setf *current-text-fields*
                          (list "New class name"))
                    (display-get-values)
                    (setf *action*
                          `(change-class-name (read-from-string ,the-class)
                                              (read-from-string (LV:value text1))))))))
```

```
(defun menu-change-class-desc ()
 (make-instance 'LV:command-menu-item
             :label "Change Class Description"
             :command
             #'(lambda () (setf *command* "change-class-desc")
               (setf *current-text-fields* (list "New Desc"))
               (display-get-values)
               (setf *action*
                     '(change-class-desc (read-from-string *class-select*)
                                          (LV:value text1)))) ))

(defun menu-validate-class ()
 (make-instance 'LV:command-menu-item
             :label "Verify the Class"
             :command
             #'(lambda () (setf *command* "validate-class")
               (verify-class (read-from-string *class-select*))
               (redraw-all-windows)) ))

(defun menu-add-attribute ()
 (make-instance 'LV:command-menu-item
             :label "Add an Attribute"
             :command
             #'(lambda () (setf *command* "add-attribute")
               (setf *current-text-fields* (list "Name"
                                                 "Desc"
                                                 "Base value"
                                                 "Lower value (opt)"
                                                 "Upper value (opt)"))
               (display-get-values)
               (setf *action*
                     '(add-attribute (read-from-string *class-select*)
                                     (read-from-string (LV:value text1))
                                     (LV:value text2)
                                     (read-from-string (LV:value text3))
                                     (if (not (equal (LV:value text4) ""))
                                        (read-from-string (LV:value text4)))
                                     (if (not (equal (LV:value text5) ""))
                                        (read-from-string (LV:value text5))) )))))

(defun menu-delete-attribute ()
 (make-instance 'LV:command-menu-item
             :label "Delete the Attribute"
             :command
             #'(lambda () (setf *command* "delete-att")
               (delete-attribute (read-from-string *class-select*)
                                 (read-from-string (second *component-select*)))
               (setf *component-select* '())
               (setf *attribute-component* '())
               (redraw-all-windows))))

(defun menu-change-att-name ()
 (make-instance 'LV:command-menu-item
             :label "Change Attribute Name"
             :command
```

```
                        #'(lambda () (setf *command* "change-att-name")
                           (let ((the-att (second *component-select*)))
                             (setf *component-select* '())
                             (setf *current-text-fields* (list "New name"))
                             (setf (LV:state att-components-menu) :inactive)
                             (display-get-values)
                             (setf *action*
                                     `(change-att-name (read-from-string *class-select*)
                                                         (read-from-string ,the-att)
                                                         (read-from-string (LV:value text1)))) ))))

(defun menu-change-att-desc ()
 (make-instance 'LV:command-menu-item
                        :label "Change Attribute Desciption"
                        :command
                        #'(lambda () (setf *command* "change-att-desc")
                           (setf *current-text-fields* (list "New desc"))
                           (display-get-values)
                           (setf *action*
                                   '(change-att-desc (read-from-string *class-select*)
                                                      (read-from-string (second *component-select*))
                                                      , .V:value text1))) )))

(defun menu-change-initial-value ()
 (make-instance 'LV:command-menu-item
                        :label "Change Initial Value"
                        :command
                        #'(lambda () (setf *command* "change-initial-value")
                           (setf *current-text-fields*
                                   (list "New initial value (must be a list)"))
                           (display-get-values)
                           (setf *action*
                                   '(change-initial-value
                                     (read-from-string *class-select*)
                                     (read-from-string (second *component-select*))
                                     (read-from-string (LV:value text1)))))))

(defun menu-change-att-a-set ()
 (make-instance 'LV:command-menu-item
                        :label "Change Attribute Legal Value"
                        :command
                        #'(lambda () (setf *command* "change-att-a-set")
                           (setf (LV:value text1) '())
                           (setf (LV:value text2) '())
                           (setf *current-text-fields* (list "Base value"
                                                              "Lower value (opt)"
                                                              "Upper value (opt)"))
                           (display-get-values)
                           (setf *action*
                                   '(change-attr-a-set
                                     (read-from-string *class-select*)
                                     (read-from-string
                                      (second *component-select*))
                                     (read-from-string (LV:value text1))
                                     (if (not (equal (LV:value text2) ""))
                                          (read-from-string (LV:value text2)))
                                   (if (not (equal (LV:value text3) ""))
                                        (read-from-string (LV:value text3))))) )))

(defun menu-validate-att ()
 (make-instance 'LV:command-menu-item
               :label "Verify the Attribute"
               :command
               #'(lambda () (setf *command* "validate-att")
                  (setf (verif (return-attribute (read-from-string *class-select*)
                                                 (read-from-string
                                                  (second *component-select*))))
                        t)
                  (redraw-all-windows)) ))

(defun menu-add-service ()
 (make-instance 'LV:command-menu-item
               :label "Add a Service"
               :command
               #'(lambda () (setf *command* "add-service")
                  (setf *current-text-fields* (list "New service name"
                                                    "New service desc"))
                  (display-get-values)
                  (setf *action*
                        '(add-service
                          (read-from-string *class-select*)
                          (read-from-string (LV:value text1))
                          (LV:value text2))))))

(defun menu-add-service-template ()
 (make-instance 'LV:command-menu-item
               :label "Add Service Using Template"
               :command
               #'(lambda () (setf *command* "add-service-template")
                  (setf *current-text-fields*
                        (list "Template (change,return,add,remove)"
                              "Attribute name"
                              "Service name"))
                  (display-get-values)
                  (setf *action*
                        '(add-template
                          (read-from-string *class-select*)
                          (read-from-string (LV:value text1))
                          (read-from-string (LV:value text2))
                          (read-from-string (LV:value text3)))))))

(defun menu-delete-service ()
 (make-instance 'LV:command-menu-item
               :label "Delete the Service"
               :command
               #'(lambda () (setf *command* "delete-serv")
                  (delete-service (read-from-string *class-select*)
                                  (read-from-string (second *component-select*)))
                  (setf *component-select* '())
                  (setf *service-component* '())
                  (redraw-all-windows))))
```

```
(defun menu-change-serv-name ()
 (make-instance 'LV:command-menu-item
                :label "Change Service Name"
                :command
                #'(lambda () (setf *command* "change-serv-name")
                   (let ((the-serv (second *component-select*)))
                     (setf *component-select* '())
                     (setf *current-text-fields* (list "New name"))
                     (setf (LV:state serv-components-menu) :inactive)
                     (display-get-values)
                     (setf *action*
                           `(change-service-name (read-from-string *class-select*)
                                                 (read-from-string ,the-serv)
                                                 (read-from-string (LV:value text1))))))))

(defun menu-change-serv-desc ()
 (make-instance 'LV:command-menu-item
                :label "Change Service Description"
                :command
                #'(lambda ()
                   (setf *command* "change-serv-desc")
                   (setf *current-text-fields* (list "New desc"))
                   (display-get-values)
                   (setf *action*
                         '(change-serv-desc (read-from-string *class-select*)
                                            (read-from-string (second *component-select*))
                                            (LV:value text1))))))

(defun menu-add-input-para ()
 (make-instance 'LV:command-menu-item
                :label "Add Input Parameter"
                :command
                #'(lambda () (setf *command* "add-input-para")
                   (setf *current-text-fields* (list "New parameter name"
                                                     "New parameter values"))
                   (display-get-values)
                   (setf *action*
                         '(change-input-set
                           (read-from-string *class-select*)
                           (read-from-string (second *component-select*))
                           '(*add)
                           (list (read-from-string (LV:value text1))
                                 (read-from-string (LV:value text2))))))))

(defun menu-remove-input-para ()
 (make-instance 'LV:command-menu-item
                :label "Remove Existing Input Parameter"
                :command
                #'(lambda () (setf *command* "remove-input-para")
                   (setf *current-text-fields* (list "Parameter name"
                                                     "Parameter value"))
                   (display-get-values)
                   (setf *action*
                         '(change-input-set
                           (read-from-string *class-select*)
                           (read-from-string (second *component-select*))
```

```
                                          (list (read-from-string (LV:value text1))
                                                (read-from-string (LV:value text2)))
                                          '(*delete))))))

(defun menu-change-input-para ()
 (make-instance 'LV:command-menu-item
                          :label "Change Existing Input Parameter"
                          :command
                          #'(lambda () (setf *command* "change-input-para")
                             (setf *current-text-fields* (list "Old parameter name"
                                                                "Old parameter value"
                                                                "New parameter name"
                                                                "New parameter value"))
                             (display-get-values)
                             (setf *action*
                                   '(change-input-set
                                     (read-from-string *class-select*)
                                     (read-from-string (second *component-select*))
                                     (list (read-from-string (LV:value text1))
                                           (read-from-string (LV:value text2)))
                                     (list (read-from-string (LV:value text3))
                                           (read-from-string (LV:value text4)))))))))

(defun menu-add-output-para ()
 (make-instance 'LV:command-menu-item
                          :label "Add Output Parameter"
                          :command
                          #'(lambda () (setf *command* "add-output-para")
                             (setf *current-text-fields* (list "New parameter value"))
                             (display-get-values)
                             (setf *action*
                                   '(change-output-set
                                     (read-from-string *class-select*)
                                     (read-from-string (second *component-select*))
                                     '*add
                                     (read-from-string (LV:value text1)))))))

(defun menu-remove-output-para ()
 (make-instance 'LV:command-menu-item
                          :label "Remove Existing Output Parameter"
                          :command
                          #'(lambda () (setf *command* "remove-output-para")
                             (setf *current-text-fields* (list "Old parameter values"))
                             (display-get-values)
                             (setf *action*
                                   '(change-output-set
                                     (read-from-string *class-select*)
                                     (read-from-string (second *component-select*))
                                     (read-from-string (LV:value text1))
                                     '*delete)))))

(defun menu-change-output-para ()
 (make-instance 'LV:command-menu-item
                          :label "Change Existing Output Parameter"
                          :command
                          #'(lambda () (setf *command* "change-output-para")
                  (setf *current-text-fields* (list "Old parameter values"
                                                    "New parameter values"))
                  (display-get-values)
                  (setf *action*
                        '(change-output-set
                          (read-from-string *class-select*)
                          (read-from-string (second *component-select*))
                          (read-from-string (LV:value text1))
                          (read-from-string (LV:value text2)))))))

(defun menu-change-pre ()
 (make-instance 'LV:command-menu-item
                :label "Change Precondition"
                :command
                #'(lambda () (setf *command* "change-pre")
                  (setf *current-text-fields* (list "New precondition"))
                  (display-get-values)
                  (setf *action*
                        '(change-serv-pre
                          (read-from-string *class-select*)
                          (read-from-string (second *component-select*))
                          (read-from-string (LV:value text1)))))))

(defun menu-add-post-atts ()
 (make-instance 'LV:command-menu-item
                :label "Add an Attribute/Value"
                :command
                #'(lambda () (setf *command* "add-post-atts")
                  (setf *current-text-fields* (list "New attribute name"
                                                    "New attribute value"))
                  (display-get-values)
                  (setf *action*
                        '(change-serv-post-atts
                          (read-from-string *class-select*)
                          (read-from-string (second *component-select*))
                          '(*add)
                          (list (read-from-string (LV:value text1))
                                (read-from-string (LV:value text2))))))))

(defun menu-remove-post-atts ()
 (make-instance 'LV:command-menu-item
                :label "Remove an Existing Attribute/Value"
                :command
                #'(lambda () (setf *command* "remove-post-atts")
                  (setf *current-text-fields* (list "Old attribute name"
                                                    "Old attribute value"))
                  (display-get-values)
                  (setf *action*
                        '(change-serv-post-atts
                          (read-from-string *class-select*)
                          (read-from-string (second *component-select*))
                          (list (read-from-string (LV:value text1))
                                (read-from-string (LV:value text2)))
                          '(*delete))))))

(defun menu-change-post-atts ()
 (make-instance 'LV:command-menu-item
                :label "Change Existing Attribute/Value"
                :command
                #'(lambda () (setf *command* "change-post-atts")
                   (setf *current-text-fields* (list "Old attribute name"
                                                     "Old attribute value"
                                                     "New attribute name"
                                                     "New attribute value"))
                   (display-get-values)
                   (setf *action*
                         '(change-serv-post-atts
                           (read-from-string *class-select*)
                           (read-from-string (second *component-select*))
                           (list (read-from-string (LV:value text1))
                                 (read-from-string (LV:value text2)))
                           (list (read-from-string (LV:value text3))
                                 (read-from-string (LV:value text4))))))))

(defun menu-add-post-mess ()
 (make-instance 'LV:command-menu-item
                :label "Add Message to Postcondition"
                :command
                #'(lambda () (setf *command* "add-post-mess")
                   (setf *current-text-fields* (list "Class name"
                                                     "Service name"))
                   (display-get-values)
                   (setf *action*
                         '(change-serv-post-mess
                           (read-from-string *class-select*)
                           (read-from-string (second *component-select*))
                           '(*add)
                           (list (read-from-string (LV:value text1))
                                 (read-from-string (LV:value text2))))))))

(defun menu-remove-post-mess ()
 (make-instance 'LV:command-menu-item
                :label "Remove Message From Postcondition"
                :command
                #'(lambda () (setf *command* "remove-post-mess")
                   (setf *current-text-fields* (list "Class name"
                                                     "Service name"))
                   (display-get-values)
                   (setf *action*
                         '(change-serv-post-mess
                           (read-from-string *class-select*)
                           (read-from-string (second *component-select*))
                           (list (read-from-string (LV:value text1))
                                 (read-from-string (LV:value text2)))
                           '(*delete))))))

(defun menu-change-post-mess ()
 (make-instance 'LV:command-menu-item
                :label "Change Existing Message in Postcondition"
                :command
                #'(lambda () (setf *command* "change-post-mess")
                   (setf *current-text-fields* (list "Old class name"
                                                "Old service name"
                                                "New class name"
                                                "New service name"))
                  (display-get-values)
                  (setf *action*
                        '(change-serv-post-mess
                          (read-from-string *class-select*)
                          (read-from-string (second *component-select*))
                          (list (read-from-string (LV:value text1))
                                (read-from-string (LV:value text2)))
                          (list (read-from-string (LV:value text3))
                                (read-from-string (LV:value text4))))))))

(defun menu-validate-serv ()
 (make-instance 'LV:command-menu-item
               :label "Verify the Service"
               :command
               #'(lambda () (setf *command* "validate-serv")
                  (setf (verif (return-service (read-from-string *class-select*)
                                              (read-from-string
                                               (second *component-select*))))
                        t)
                  (redraw-all-windows)) ))

(defun menu-add-wp-comp ()
 (make-instance 'LV:command-menu-item
               :label "Add Whole/Part Relation"
               :command
               #'(lambda () (setf *command* "add-wp-comp")
                  (setf *current-text-fields* (list "Class1"
                                                "Range1"
                                                "Class2"
                                                "Range2"))
                  (display-get-values)
                  (setf *action*
                        '(add-new-relation
                          (read-from-string *class-select*)
                          (make-relation
                           :class1 (read-from-string (LV:value text1))
                           :range1 (read-from-string (LV:value text2))
                           :class2 (read-from-string (LV:value text3))
                           :range2 (read-from-string (LV:value text4))))))))

(defun menu-remove-wp-comp ()
 (make-instance 'LV:command-menu-item
               :label "Remove Existing Whole/Part Relation"
               :command
               #'(lambda () (setf *command* "remove-wp-comp")
                  (setf *current-text-fields* (list "Class1"
                                                "Range1"
                                                "Class2"
                                                "Range2"))
                  (display-get-values)
                  (setf *action*
                        '(delete-relation
                          (read-from-string *class-select*)
```

```
                      (make-relation
                       :name 'whole/part
                       :class1 (read-from-string (LV:value text1))
                       :range1 (read-from-string (LV:value text2))
                       :class2 (read-from-string (LV:value text3))
                       :range2 (read-from-string (LV:value text4))))))))

(defun menu-change-ranges ()
 (make-instance 'LV:command-menu-item
              :label "Change Ranges"
              :command
              #'(lambda () (setf *command* "range")
                (setf *current-text-fields* (list "Old relation name"
                                                  "Class1"
                                                  "Range1"
                                                  "Class2"
                                                  "Range2"
                                                  "New range1"
                                                  "New range2"))
                (display-get-values)
                (setf *action*
                     '(change-relation-range
                      (read-from-string *class-select*)
                      (make-relation
                       :name (read-from-string (LV:value text1))
                       :class1 (read-from-string (LV:value text2))
                       :range1 (read-from-string (LV:value text3))
                       :class2 (read-from-string (LV:value text4))
                       :range2 (read-from-string (LV:value text5)))
                      (read-from-string (LV:value text6))
                      (read-from-string (LV:value text7)))))))

(defun menu-other-class ()
 (make-instance 'LV:command-menu-item
              :label "Change Other Class"
              :command
              #'(lambda () (setf *command* "other-class")
                (setf *current-text-fields* (list "Relation name"
                                                  "Class1"
                                                  "Range1"
                                                  "Class2"
                                                  "Range2"
                                                  "New other class"))
                (display-get-values)
                (setf *action*
                     '(change-relation-class
                      (read-from-string *class-select*)
                      (make-relation
                       :name (read-from-string (LV:value text1))
                       :class1 (read-from-string (LV:value text2))
                       :range1 (read-from-string (LV:value text3))
                       :class2 (read-from-string (LV:value text4))
                       :range2 (read-from-string (LV:value text5)))
                      (read-from-string (LV:value text6)))))))
```

```
(defun menu-add-rel-comp ()
 (make-instance 'LV:command-menu-item
             :label "Add an Other Relation"
             :command
             #'(lambda () (setf *command* "add-rel-comp")
                (setf *current-text-fields* (list "Relation name"
                                                  "Class1"
                                                  "Range1"
                                                  "Class2"
                                                  "Range2"))
                (display-get-values)
                (setf *action*
                      '(add-new-relation
                        (read-from-string *class-select*)
                        (make-relation
                         :name (read-from-string (LV:value text1))
                         :class1 (read-from-string (LV:value text2))
                         :range1 (read-from-string (LV:value text3))
                         :class2 (read-from-string (LV:value text4))
                         :range2 (read-from-string (LV:value text5))))))))

(defun menu-remove-rel-comp ()
 (make-instance 'LV:command-menu-item
             :label "Remove an Other Relation"
             :command
             #'(lambda () (setf *command* "remove-rel-comp")
                (setf *current-text-fields* (list "Relation name"
                                                  "Class1"
                                                  "Range1"
                                                  "Class2"
                                                  "Range2"))
                (display-get-values)
                (setf *action*
                      '(delete-relation
                        (read-from-string *class-select*)
                        (make-relation
                         :name (read-from-string (LV:value text1))
                         :class1 (read-from-string (LV:value text2))
                         :range1 (read-from-string (LV:value text3))
                         :class2 (read-from-string (LV:value text4))
                         :range2 (read-from-string (LV:value text5))))))))

(defun menu-rel-name ()
 (make-instance 'LV:command-menu-item
             :label "Change Relation Name"
             :command
             #'(lambda () (setf *command* "rel-name")
                (setf *current-text-fields* (list "Relation name"
                                                  "Class1"
                                                  "Range1"
                                                  "Class2"
                                                  "Range2"
                                                  "New relation name"))
                (display-get-values)
                (setf *action*
                      '(change-relation-name
                        (read-from-string *class-select*)
                        (make-relation
                         :name (read-from-string (LV:value text1))
                         :class1 (read-from-string (LV:value text2))
                         :range1 (read-from-string (LV:value text3))
                         :class2 (read-from-string (LV:value text4))
                         :range2 (read-from-string (LV:value text5)))
                        (read-from-string (LV:value text6)))))))

(defun menu-add-parent ()
 (make-instance 'LV:command-menu-item
            :label "Add a Parent"
            :command
            #'(lambda () (setf *command* "add-parent")
              (setf *current-text-fields*
                   (list "Parent to be added"))
              (display-get-values)
              (setf *action*
                   '(add-parent
                    (read-from-string *class-select*)
                    (read-from-string (LV:value text1)))))))

(defun menu-remove-parent ()
 (make-instance 'LV:command-menu-item
            :label "Remove a Parent"
            :command
            #'(lambda () (setf *command* "remove-parent")
              (setf *current-text-fields*
                   (list "Parent to be removed"))
              (display-get-values)
              (setf *action*
                   '(remove-parent
                    (read-from-string *class-select*)
                    (read-from-string (LV:value text1)))))))

(defun menu-change-parent ()
 (make-instance 'LV:command-menu-item
            :label "Change an Existing Parent"
            :command
            #'(lambda () (setf *command* "change-parent")
              (setf *current-text-fields*
                   (list "Parent to be changed"
                        "New parent"))
              (display-get-values)
              (setf *action*
                   '(change-parent
                    (read-from-string *class-select*)
                    (read-from-string (LV:value text1))
                    (read-from-string (LV:value text2)))))))

;;________________________________________________________________________
;; The following creates the menu choices for each component in the model.
;; The menu choice is based on the values of *class-select*,
;; *component-select*, *attribute-component* and *service-component*
;; For example, if *class-select* is null, no class is selected and the
;; only modification choices is to add a new class.
```

```
;;___________________________________________________________________________
(defun get-choice-menu ()
 (let* ((add-class (menu-add-class))
        (delete-class (menu-delete-class))
        (change-class-name (menu-change-class-name))
        (change-class-desc (menu-change-class-desc))
        (validate-class (menu-validate-class))
        (add-attribute (menu-add-attribute))
        (delete-attribute (menu-delete-attribute))
        (change-att-name (menu-change-att-name))
        (change-att-desc (menu-change-att-desc))
        (change-initial-value (menu-change-initial-value))
        (change-att-a-set (menu-change-att-a-set))
        (validate-att (menu-validate-att))
        (add-service (menu-add-service))
        (add-service-template (menu-add-service-template))
        (delete-service (menu-delete-service))
        (change-serv-name (menu-change-serv-name))
        (change-serv-desc (menu-change-serv-desc))
        (add-input-para (menu-add-input-para))
        (remove-input-para (menu-remove-input-para))
        (change-input-para (menu-change-input-para))
        (add-output-para (menu-add-output-para))
        (remove-output-para (menu-remove-output-para))
        (change-output-para (menu-change-output-para))
        (change-pre (menu-change-pre))
        (add-post-atts (menu-add-post-atts))
        (remove-post-atts (menu-remove-post-atts))
        (change-post-atts (menu-change-post-atts))
        (add-post-mess (menu-add-post-mess))
        (remove-post-mess (menu-remove-post-mess))
        (change-post-mess (menu-change-post-mess))
        (validate-serv (menu-validate-serv))
        (add-wp-comp (menu-add-wp-comp))
        (remove-wp-comp (menu-remove-wp-comp))
        (change-ranges (menu-change-ranges))
        (other-class (menu-other-class))
        (add-rel-comp (menu-add-rel-comp))
        (remove-rel-comp (menu-remove-rel-comp))
        (rel-name (menu-rel-name))
        (add-parent (menu-add-parent))
        (remove-parent (menu-remove-parent))
        (change-parent (menu-change-parent))
        ;; The following are the menus.  Above were the menu components.
        (entire-model (list add-class))
        (entire-class
         (list delete-class validate-class add-attribute add-service add-service-template))
        (class-name (list change-class-name))
        (class-desc (list change-class-desc))
        (entire-att (list delete-attribute validate-att))
        (att-name (list change-att-name))
        (att-desc (list change-att-desc))
        (att-ini (list change-initial-value))
        (att-a-set (list change-att-a-set))
        (entire-serv (list delete-service validate-serv))
        (serv-name (list change-serv-name))
```

```
            (serv-desc (list change-serv-desc))
            (serv-input (list add-input-para remove-input-para change-input-para))
            (serv-output (list add-output-para remove-output-para change-output-para))
            (serv-pre (list change-pre))
            (serv-post-atts (list add-post-atts remove-post-atts change-post-atts))
            (serv-post-mess (list add-post-mess remove-post-mess change-post-mess))
            (wp (list add-wp-comp remove-wp-comp change-ranges other-class))
            (rel (list add-rel-comp remove-rel-comp change-ranges other-class rel-name))
            (inh (list add-parent remove-parent change-parent)))
    (cond ((not *class-select*) entire-model)
          ((equal *component-select* '()) entire-class)
          (*component-select*
           (cond ((equal *component-select* "name") class-name)
                 ((equal *component-select* "desc") class-desc)
                 ((equal *component-select* "w-p") wp)
                 ((equal *component-select* "rel") rel)
                 ((equal *component-select* "inh") inh)
                 ((and (listp *component-select*)
                       (equal (first *component-select*) 'attribute))
                  (cond ((not *attribute-component*) entire-att)
                        ((equal *attribute-component* "name") att-name)
                        ((equal *attribute-component* "desc") att-desc)
                        ((equal *attribute-component* "ini") att-ini)
                        ((equal *attribute-component* "a-set") att-a-set)))
                 ((and (listp *component-select*)
                       (equal (first *component-select*) 'service))
                  (cond ((not *service-component*) entire-serv)
                        ((equal *service-component* "name") serv-name)
                        ((equal *service-component* "desc") serv-desc)
                        ((equal *service-component* "input") serv-input)
                        ((equal *service-component* "output") serv-output)
                        ((equal *service-component* "pre") serv-pre)
                        ((equal *service-component* "post-atts") serv-post-atts)
                        ((equal *service-component* "post-mess") serv-post-mess))) )))))

(setf command-menu
   (make-instance 'LV:menu
                  :choices #'get-choice-menu))

(setf command-select
   (make-instance 'LV:menu-button
                  :parent *oaks-panel*
                  :label "Action"
                  :menu command-menu))

;;*******************************************************************************
;;
;; Create a button on the main oaks panel that will display the advisory-issues
;; in the current-component-vp when pushed.  The window is changed to another
;; component whenever any other component is selected.
;;*******************************************************************************
;;

;;_____________________________________________________________________________
;; The function that displays *advisory-issues*
;;_____________________________________________________________________________
(defun display-advisory-issues ()
  (advisory-tests)
```

```
 (let ((x 20)
       (y 20))
  (LV:clear current-component-vp)
  (LV:draw-string current-component-vp x y "ADVISORY ISSUES")
  (setf y (+ 25 y))
  (dolist (one-entry *advisory-issues*)
        (LV:draw-string current-component-vp x y (princ-to-string one-entry))
        (setf y (+ 15 y))) ))

;;________________________________________________________________________
;; Create the advisory-issues button
;;________________________________________________________________________
(setf *advisory-button*
   (make-instance 'LV:command-button
                  :parent *oaks-panel*
                  :label "Advisory Issues"
                  :command #'display-advisory-issues))

;;**************************************************************************
;; Create a button on the main oaks panel that will save the current state of the
;; *list-of-classes* and the *pending-issues* list to a file that is called each
;; time OAKS is used.
;;**************************************************************************
(setf *save-button*
   (make-instance 'LV:command-button
                  :parent *oaks-panel*
                  :label "Save"
                  :command #'write-data))

;;**************************************************************************
;; Make a pop-up window that gets any information required from the user
;; for model modifications. For example, the user does not have to provide
;; any information to delete a selected class. To change a class name,
;; the user must provide the new class name. The new name is gathered through
;; a pop-up box.
;;**************************************************************************

;;________________________________________________________________________
;; The text fields and labels that are displayed on the pop-up window
;;________________________________________________________________________
(setf *current-text-fields* '())

;;________________________________________________________________________
;; Removes the pop-up menu and conducts any action required. This action is
;; stored in *action*.
;;________________________________________________________________________
(defun remove-menu ()
 (setf (LV:mapped *done-command*) '())
 (setf text-list (list text1 text2 text3 text4 text5 text6 text7))
 (dolist (one-field *current-text-fields*)
       (setf (LV:mapped (first text-list)) '())
       (setf text-list (rest text-list)))
 (setf (LV:mapped *pop-up-panel*) '())
 (setf (LV:mapped *pop-up*) '())
 (eval *action*)
 (redraw-all-windows))
```

```
;;______________________________________________________________________
;; Displays the pop-up menu.  Depending on how much info is required,
;; one or more text field will be displayed.  How many text fields are
;; needed and the lables for each text field are contained in *current-text-fields*
;;______________________________________________________________________
(defun display-get-values ()
 (setf *pop-up* (make-instance 'LV:base-window
                               :width 550
                               :height 225
                               :left 325
                               :top 430
                               :label "User Information"
                               :mapped t))
 (let ((br (LV:bounding-region *pop-up*)))
  (setf *pop-up-panel* (make-instance 'LV:panel
                                      :parent *pop-up*
                                      :width (LV:region-width br)
                                      :height (LV:region-height br)
                                      :mapped t)))
 (setf text1 (make-instance 'LV:text-field
                            :parent *pop-up-panel*
                            :label "Text1"
                            :mapped '()))
 (setf text2 (make-instance 'LV:text-field
                            :parent *pop-up-panel*
                            :label "Text2"
                            :mapped '()))
 (setf text3 (make-instance 'LV:text-field
                            :parent *pop-up-panel*
                            :label "Text3"
                            :mapped '()))
 (setf text4 (make-instance 'LV:text-field
                            :parent *pop-up-panel*
                            :label "Text4"
                            :mapped'()))
 (setf text5 (make-instance 'LV:text-field
                            :parent *pop-up-panel*
                            :label "Text5"
                            :mapped '()))
 (setf text6 (make-instance 'LV:text-field
                            :parent *pop-up-panel*
                            :label "Text6"
                            :mapped '()))
 (setf text7 (make-instance 'LV:text-field
                            :parent *pop-up-panel*
                            :label "Text7"
                            :mapped '()))
 (setf text-list (list text1 text2 text3 text4 text5 text6 text7))
 ;; Creates the "done" button on the menu.  When it is pressed, the menu is
 ;; removed and the action required is done.
 (setf *done-command* (make-instance 'LV:command-button
                                     :parent *pop-up-panel*
                                     :label "Done"
                                     :command #'remove-menu
                                     :mapped t))
 (let ((current-text text-list))
  (dolist (one-field *current-text-fields*)
            (setf (LV:label (first current-text)) one-field)
            (setf (LV:mapped (first current-text)) t)
            (setf current-text (rest current-text))))
 (setf (LV:mapped *done-command*) t))

;;*****************************************************************************
;; Redraws all the windows.  Used whenever a change is made.
;;*****************************************************************************
(defun redraw-all-windows ()
 (create-class-list)
 (display-current-component)
 (display-pending-issues))

;;*****************************************************************************
;; Draws lines to a certain width
;;*****************************************************************************
(defun print-restricted-width (width str output-area x y)
 (let ((one-line (make-string (+ 1 width) :initial-element #\Space))
         (start-pos 0)
         (end-pos width)
         (string-length (- (array-total-size str) 1)))
   (loop
    (if (or (< string-length end-pos) (= string-length end-pos))
          (progn
           (do ((string-pos start-pos (+ string-pos 1))
                 (line-pos 0 (+ line-pos 1)))
               ((eql string-pos (+ 1 string-length)) '())
               (setf (char one-line line-pos) (char str string-pos)))
           (LV:draw-string output-area x y one-line)
           (return (+ 15 y)))
     (loop
        (if (or (char= #\Newline (char str end-pos))
                  (char= #\Space (char str end-pos)))
            (progn
             (do ((string-pos start-pos (+ string-pos 1))
                    (line-pos 0 (+ 1 line-pos)))
                   ((eql string-pos (+ 1 end-pos)) '())
                   (setf (char one-line line-pos) (char str string-pos)))
             (LV:draw-string output-area x y one-line)
             (setf one-line (make-string (+ 1 width) :initial-element #\Space))
             (setf y (+ 15 y))
             (setf start-pos (+ 1 end-pos))
             (setf end-pos (+ end-pos width))
             (return))
          (setf end-pos (- end-pos 1)) ))))))

;;*****************************************************************************
;; Create a message window that is used to display error messages to the user.
;; These error messages occur when a change is made to the model.
;;*****************************************************************************

(defun create-error-message (the-message)
 (setf *error-message-window*
         (make-instance 'LV:popup-window
                    :width 550
                    :height 40
                    :left 350
                    :top 500
                    :label "Information Message"
                    :mapped 't))
(let ((br (LV:bounding-region *error-message-window*)))
  (setf *error-message-panel* (make-instance 'LV:panel
                                        :parent *error-message-window*
                                        :width (LV:region-width br)
                                        :height (LV:region-height br)
                                        :mapped t)))
(setf *error-message*
      (make-instance 'LV:message
                    :parent *error-message-panel*
                    :label the-message
                    :mapped t)))
```

OAKS.LISP

Code to Load OAKS Environment

```
(in-package 'oaks)

;;**********************************************************************************
;; This file is loaded by the user before each session.
;;  read-data reads the information form the file "userfil". This either
;; contains the domain mode, if the user has not used OAKS yet, or the last
;; saved modified model.
;;*********************************************************************************

(load "oaksd.lisp")
(load "oaksno.lisp")
(load "oaksmod.lisp")
(load "oaksave.lisp")
(read-data)
(load "oaksui.lisp")
(in-package 'oaks)
```

REPORT DOCUMENTATION PAGE

	2. REPORT DATE September 1993	3. REPORT TYPE AND DATES COVERED Technical Report

4. TITLE AND SUBTITLE Appendix C: OAKS Code	5. FUNDING NUMBERS
6. AUTHOR(S) Nancy L. Crowley, Major, USAF	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583	8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/EN/TR/93-07
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Distribution Unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT

This technical report is an appendix to AFIT Doctoral Dissertation AFIT/DS/ENG/93-11, "On the Automation of Object-Oriented Requirements Analysis", September 1993. The technical report contains the LISP code for the Object-oriented Requirements Analysis (OORA) Automated Knowledge System (OAKS).

14. SUBJECT TERMS Object-Oriented,Requirements	15. NUMBER OF PAGES 155
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL